

# Verifying Finite-State Safety Properties on Millions of Lines of Code

Suhabe Bugrara and Alex Aiken

Stanford University  
{suhabe,aiken}@stanford.edu

**Abstract.** We present a context-sensitive, flow-sensitive, field-sensitive, and intraprocedurally path-sensitive static analysis capable of verifying finite-state safety properties of very large systems. Unusually for finite-state property verifiers, our system analyzes functions separately, and it is this feature that enables scalability. We evaluate an implementation of our analysis by trying to verify the absence of unchecked, untrusted pointer dereferences in the entire Linux operating system with over 6.2 million lines of code. Our system has a 1.8% false positive rate and fails to analyze 0.17% of all procedures.

## 1 Introduction

Finite-state safety properties are an important class of specifications [12, 6, 5, 7, 2, 10, 11, 13, 8]. In this paper, we describe a system for analyzing finite-state safety properties that is sound, scales to the largest systems available to us (the Linux OS, with over 6.2 million lines of code), and has a low false positive rate (1.8% in the application we studied). We believe ours is the first system to demonstrate this level of scalability while maintaining soundness and precision.

The key to scalability in our system, and an unusual feature compared to many contemporary program analysis systems, is that each procedure  $f$  is analyzed separately and only information about  $f$ 's *summary*, which captures  $f$ 's behavior with respect to the finite-state property, is communicated to other procedures that use  $f$ . Analyzing a procedure in isolation makes it feasible to apply relatively expensive techniques: our analysis is context-sensitive, flow-sensitive, field-sensitive, and intraprocedurally path-sensitive, and this combination of features is what keeps the false positive rate low. Procedure summaries abstract the intraprocedural analysis but cause relatively little loss of information in practice, as programmers already naturally abstract at procedure boundaries. The combination of precision and scalability makes verification of complex safety properties feasible on large, complex systems.

The key to precision in our system is that every important fact is associated with a formula, or *guard*, giving the precise condition (e.g., path-sensitive condition) under which the fact holds. In our framework the conditions are boolean constraints, and we illustrate by example how a series of such guarded analyses can be composed into a complete analysis.

We have implemented our analysis and evaluated its precision and scalability by trying to verify an important security property of the entire Linux 2.6.17.1 operating system. The security property entails proving the absence of *unchecked, untrusted pointer dereferences*, described further in Section 1.1. Our implementation has a 1.8% false positive rate and fails (times out) on 0.17% of procedures.

Our soundness claim assumes several things about the programs being analyzed, the most important of which is that the program is memory safe; i.e., we do not detect errors that may arise from exploitation of buffer overruns and similar unsafe memory operations in C. We also assume that the program is single-threaded, and we do not analyze inline assembly. Finally, our analysis fails to analyze some procedures, which means that any errors in those procedures are missed, as well as potentially missing errors in other procedures that depend on the unanalyzed procedures.

## 1.1 Unchecked, Untrusted Pointer Dereferences

Operating systems divide virtual memory addresses into two regions: *kernel space* (the memory in which the kernel maintains its data structures and other state), and *user space* (the memory holding data for user processes). Any pointer created by a user process is an *untrusted pointer*. Because user processes need the kernel to perform operations on their behalf (e.g., read and write files, open network connections, etc.), untrusted pointers are often passed as arguments to kernel system calls. The potential problem occurs when the kernel dereferences an untrusted pointer: the operating system must first check that it points into user space. If an attacker creates an untrusted pointer to kernel memory, and the operating system dereferences it without checking that it points into user space first, then the attacker

could take control of the operating system by overwriting kernel data structures, read sensitive data out of kernel memory, or simply crash the machine by corrupting kernel data [12].

Figure 1 gives an example of how an untrusted pointer is checked before being dereferenced. The example consists of two procedure: `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 (i.e., is available to user applications). Consequently, its pointer parameter `u` is an untrusted pointer. Line 5 applies `access_ok` to `u` to confirm that `u` points into user space. Subsequently, line 7 calls `get` with `u` on the condition that the return value of the call to `access_ok` is non-zero which implies that the check on line 4 succeeded. Procedure `get` dereferences its pointer parameter.

The rest of this paper is organized as follows. Section 2 defines the language used in our examples and algorithms. Section 3 describes the memory model that handles pointer aliasing. Section 4 gives an overview of the safety analysis.

```

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: }

```

**Fig. 1.** Running example.

Section 5 and Section 6 describe the intraprocedural and interprocedural components of the finite-state safety analysis. Section 7 describes a technique that improves scalability. Finally, Section 8 presents experimental results.

## 2 Language

We briefly define a simple imperative language used to present our analysis. A program is a set of *procedures*.

$$\begin{aligned}
 \textit{procedure} &::= \textit{type } P \textit{ (type } v) \{ \textit{statement} \} \\
 \textit{statement} &::= \textit{type } v \mid \textit{return } v \mid v := \mathbb{Z} \mid v_1 := *v_2 \mid v_1 := \mathbb{Q} (v_2) \\
 &\quad \mid \textit{statement} ; \textit{statement} \mid \textit{if } (v \neq 0) \textit{ statement } \textit{else } \textit{statement} \\
 \textit{type} &::= \textit{void} \mid \textit{int} \mid \textit{int*} \mid \textit{int**}
 \end{aligned}$$

A procedure has a return type, a single formal parameter, and a statement; the statement forms are self explanatory. Our analysis always takes place in some procedure, thus we superscript domains 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  $v_i$  of procedure  $P \in Proc$ .

## 3 Memory Model

Our safety analysis uses the Saturn alias analysis [9]. This section describes the interface of the memory model used by the safety analysis; 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 total, injective function  $varloc^P \in Var^P \rightarrow Loc^P$  assigns each variable  $v \in Var^P$  to a location  $l \in Loc^P$ . A set of *abstract atomic predicates*  $Pred^P$  consists of predicates over locations. The set

of *abstract guards*  $Guard^P$  are the usual propositional formulas ( $\wedge$ ,  $\vee$ ,  $\neg$ ) over  $Pred^P$  and a finite set of boolean variables  $BoolVar^P$ .

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

### 3.1 Guarded Points-to Graphs

A *guarded points-to graph*  $\rho \in PointsTo^P = Loc^P \times Loc^P \rightarrow Guard^P$  is a total function that associates a guard  $\varphi \in Guard^P$  with each pair of locations  $l_i, l_j \in Loc^P$ , representing the condition under which  $l_i$  points to  $l_j$ . The *rvalue evaluation* function  $rval^P \in PointsTo^P \rightarrow (Rvalue^P \times Loc^P) \rightarrow Guard^P$  gives the guard under which an rvalue *points-to* a given location. Similarly, the *lvalue evaluation* function  $lval^P \in PointsTo^P \rightarrow (Lvalue^P \times Loc^P) \rightarrow Guard^P$  gives the guard under which an lvalue is *represented* by a given location.

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

### 3.2 Location Instantiation

The abstract locations  $Loc^P$  are disjoint from  $Loc^Q$  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  $\rho \in PointsTo^P$  be the points-to graph encoding the points-to information at `s`. The *location instantiation* function  $\mathcal{I}_{Loc}^P \in PointsTo^P \rightarrow (Loc^Q \times Loc^P) \rightarrow Guard^P$  gives the guard under which the abstract locations  $l_1^Q \in Loc^Q$  and  $l_2^P \in Loc^P$  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  $\rho_{11} \in PointsTo^{\text{get}}$  be the points-to graph at entry to procedure `get` on line 11 and  $\rho_7 \in PointsTo^{\text{syscall}}$  be the points-to graph at the call statement on line 7. Also let  $l_u, l_{*u} \in Loc^{\text{syscall}}$  and  $l_y, l_{*y} \in Loc^{\text{get}}$ , where  $varloc^{\text{syscall}}(u) = l_u$  and  $varloc^{\text{get}}(y) = l_y$ . The locations  $l_u$  and  $l_y$  never correspond to the same set

of concrete locations because the concrete locations represented by  $l_u$  are allocated on the stack of `syscall` while the concrete locations represented by  $l_y$  are allocated on the stack of `get`. Thus,  $\mathcal{I}_{Loc}^{syscall}(\rho_\tau)(l_y, l_u) = false$ . However,  $l_{*u}$  and  $l_{*y}$  may correspond to the same set of concrete locations because of the implicit pointer copy that occurs from  $l_u$  to  $l_y$  at the call statement. Thus,  $\mathcal{I}_{Loc}^{syscall}(\rho_\tau)(l_{*y}, l_{*u}) = \rho_\tau(l_u, l_{*u})$  when  $\rho_{11}(l_y, l_{*y}) = true$ .  $\square$

### 3.3 Judgments

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

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

where  $\phi \in Guard$  is the intraprocedural guard under which statement  $\mathbf{s}$  is reachable,  $\phi'$  is the guard under which the point after  $\mathbf{s}$  is reachable,  $\rho \in PointsTo$  is the points-to graph at the point before  $\mathbf{s}$ ,  $\rho'$  is the points-to graph at the point after  $\mathbf{s}$ , and  $\psi$  is the guard under which transfer of control occurs across  $\mathbf{s}$ .

## 4 Safety

The finite-state safety analysis analyzes each procedure in isolation using the memory model abstraction to generate a summary *state environment* that encodes the behavior of the procedure with respect to the finite-state property. Section 5 explains the intraprocedural analysis used to generate the summary state environment of a procedure. Section 6 explains the interprocedural analysis where the summary state environment of a callee procedure is applied at a call statement. The analysis repeatedly generates and applies summary state environments until a fixed point of summaries is reached [4].

### 4.1 State Environments

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

Consider the unchecked, untrusted pointer dereference property. The set of abstract states is location-typestate pairs  $State^P = Loc^P \times Typestate$  where  $Typestate = \{\mathbf{untrusted}, \mathbf{unchecked}, \mathbf{unsafe}\}$ . The state  $(l, \mathbf{untrusted}) \in State^P$  signifies that location  $l \in Loc^P$  is an untrusted location. Similarly, the state  $(l, \mathbf{unchecked})$  signifies that  $l$  has not been checked, and the state  $(l, \mathbf{unsafe})$  signifies that  $l$  is an untrusted location that has been the target of a dereference while unchecked. Note that a location may be both **untrusted** and **unchecked**—these states are not mutually exclusive.

The summary of a procedure is generated via judgments of the form

$$\phi, \rho, \Gamma \vdash_{safety}^i \mathbf{s} : \Gamma'$$

where  $\phi \in Guard$  is the guard under which  $s \in Stmt$  is executed,  $\rho \in PointsTo$  is the points-to graph before  $\mathbf{s}$ , and  $\Gamma, \Gamma' \in StateEnv$  are the state environments before and after  $\mathbf{s}$ . The superscript  $i$  on the turnstile signifies that the judgment holds for the  $i$ th iteration of the safety analysis over the program. A fixed point of summaries is reached when for all procedures  $P$ ,  $\Gamma_i^P \cong \Gamma_{i-1}^P$ .

## 5 Intraprocedural Analysis

In this section we present the intraprocedural analysis of a procedure.

### 5.1 Procedures

The initial summary of procedure  $P$  on the 0th iteration is  $\perp$ :

$$\vdash_{safety}^0 \mathbf{t}_1 \text{ P } (\mathbf{t}_2 \text{ v}) \{ \mathbf{s} \} : \perp$$

For the  $i$ th iteration of analysis of a procedure, the safety analysis generates a *final summary*  $\Gamma_{sum}$  by abstracting information away (via  $\alpha$ ) from a more precise *preliminary summary*  $\Gamma_{prelim}$ .

$$\frac{\phi_{init}, \rho_{init}, \Gamma_{init} \vdash_{safety}^i \mathbf{s} : \Gamma_{prelim} \quad \Gamma_{sum} = \alpha(\Gamma_{prelim})}{\vdash_{safety}^i \mathbf{t}_1 \text{ P } (\mathbf{t}_2 \text{ v}) \{ \mathbf{s} \} : \Gamma_{sum}}$$

Now, procedure  $P$  may be called in many different contexts, and so we want one state environment  $\Gamma_{init}$  that polymorphically represents the state environment in all those calling contexts. Thus, the initial state environment uses a fresh boolean variable called a *context variable* to represent the guard of a state in any calling context. Formally, let the context variables  $XVar^P \subseteq BoolVar^P$  be a set of boolean variables of cardinality  $|State^P|$ . The polymorphic initial state environment  $xvar^P \in State^P \rightarrow XVar^P$  assigns each state a context variable.

The polymorphic initial state environment does not incorporate any information about the guard for a state in calling contexts, allowing the safety analysis to generate fully polymorphic summaries that are not specialized to the program's calling contexts. Section 7 describes another choice for the initial state environment incorporating information about the satisfiability of the guard for a state in calling contexts. This additional information allows the safety analysis to compute more concise summaries, which is important for the scalability of some analyses, including the unchecked, untrusted pointer dereference analysis.

The *summary abstraction* function  $\alpha \in StateEnv \rightarrow StateEnv$  computes a conservative approximation  $\alpha(\Gamma)$  of a state environment  $\Gamma$  such that  $\Gamma \sqsubseteq \alpha(\Gamma)$ . The *context variable abstraction*  $\alpha_{XVar}$  approximates a state environment by the

most precise state environment whose guards only contain context variables. The context variable abstraction  $\Gamma' = \alpha_{XVar}(\Gamma)$  is characterized by  $\Gamma \sqsubseteq \Gamma'$  and  $(\Gamma \sqsubseteq \Gamma'' \sqsubseteq \Gamma') \Rightarrow (\Gamma'' \cong \Gamma')$  such that  $atoms(\Gamma') \subseteq XVar$  and  $atoms(\Gamma'') \subseteq XVar$  for any  $\Gamma''$ . 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 eliminates intraprocedural path-sensitivity while retaining all context variables.

## 5.2 Sequences

The following inference rule describes how sequences are handled.

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

The first antecedent uses the  $\vdash_{mem}$  judgment to signify that the safety analysis uses the memory model to compute the resulting statement guard  $\phi'$  and points-to graph  $\rho'$  after statement  $\mathbf{s}_1$ . The middle antecedent signifies that the safety analysis uses the statement guard  $\phi$  and points-to graph  $\rho$  to compute the resulting state environment  $\Gamma_1$  after executing  $\mathbf{s}_1$ . Similarly, the last antecedent signifies that the safety analysis uses the statement guard  $\phi'$  and points-to graph  $\rho'$  to compute the final state environment  $\Gamma_2$ .

## 5.3 Branches

The following inference rule describes how branches are handled.

$$\frac{\begin{array}{l} \phi, \rho \vdash_{mem} \mathbf{s}_1 : \phi_1, \rho_1, \psi_1 \\ \phi, \rho \vdash_{mem} \mathbf{s}_2 : \phi_2, \rho_2, \psi_2 \\ \phi, \rho, \Gamma \vdash_{safety}^i \mathbf{s}_1 : \Gamma_1 \\ \phi, \rho, \Gamma \vdash_{safety}^i \mathbf{s}_2 : \Gamma_2 \end{array}}{\phi, \rho, \Gamma \vdash_{safety}^i \text{if } (v \neq 0) \{ \mathbf{s}_1 \} \text{ else } \{ \mathbf{s}_2 \} : \Gamma'} \quad \Gamma' = \text{refine}(\Gamma_1, \psi_1) \sqcup \text{refine}(\Gamma_2, \psi_2)$$

where  $\text{refine} \in (\text{StateEnv} \times \text{Guard}) \rightarrow \text{StateEnv}$  be a function that refines a state environment  $\Gamma$  with a guard  $\varphi$  by conjoining the guard for each state in  $\Gamma$  with  $\varphi$  as in  $\text{refine}(\Gamma, \varphi)(q) = \Gamma(q) \wedge \varphi$ .

## 5.4 Unchecked, Untrusted Pointer Dereferences

Rules specific to the unchecked, untrusted pointer dereference property are given with judgment of the form

$$\phi, \rho, \Gamma \vdash_{untrusted}^i \mathbf{s} : \Gamma'$$

The components have the same meaning as in the  $\vdash_{safety}$  judgment (Section 4).

**System Call Initial State Environment** An important characteristic of the unchecked, untrusted pointer dereference property is that any location reachable from an untrusted pointer is also an untrusted pointer. When analyzing a system call  $P$ , a special initial state environment guards each  $(l, \text{untrusted})$  state with  $true$ , where  $l$  is a location reachable from the formal parameter  $\mathbf{u}$  of the system call via a series of dereferences. Let  $\text{Syscall}$  be the set of system call procedures. Formally, the initial state environment  $\Gamma_{init}^P$  where  $P \in \text{Syscall}$  is

$$\Gamma_{init}^P(q) = \begin{cases} \text{reachable}(\rho_{init}^P)(\text{varloc}^P(\mathbf{u}), l) & q = (l, \text{untrusted}) \\ true & q = (l, \text{unchecked}) \\ false & q = (l, \text{unsafe}) \end{cases}$$

where  $\text{reachable} \in \text{PointsTo} \rightarrow (\text{Loc} \times \text{Loc}) \rightarrow \text{Guard}$  gives the guard under which a location is reachable via a series of dereferences from another location, recursively defined by

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

**Checking** Let `access_ok` be a procedure that returns a nonzero value if and only if `access_ok`'s untrusted pointer argument points into user space. Consider the statement  $\mathbf{v}_1 := \text{access\_ok}(\mathbf{v}_2)$ . A location  $l$  is `unchecked` after the call if  $l$  is `unchecked` before the call and the call does not check  $l$ . Thus, if  $\Gamma$  is the state environment before the call, then  $l$  is `unchecked` after the call if  $\Gamma(l, \text{unchecked})$  and either  $\mathbf{v}_2$  does not point to  $l$  or  $\mathbf{v}_1 = 0$  after the call. Now,  $\varphi_1 \equiv \neg(\text{rval}(\rho)(\mathbf{v}_2, l) \wedge \phi)$  is the guard under which  $\mathbf{v}_2$  does not point to  $l$ , and  $\varphi_2 \equiv \bigvee_{l'} [\text{rval}(\rho)(\mathbf{v}_1, l') \wedge (l' = 0)] \wedge \phi$  is the the guard under which  $\mathbf{v}_1 = 0$ , where  $\rho$  is the points-to graph at the call statement and  $\phi$  is the guard under which the call statement executes. The following inference rule describes how the analysis updates the state environment with the guard where  $l$  remains `unchecked`.

$$\frac{\begin{array}{c} l \in \text{Loc} \\ \varphi_1 \equiv \neg(\text{rval}(\rho)(\mathbf{v}_2, l) \wedge \phi) \\ \varphi_2 \equiv \bigvee_{l'} [\text{rval}(\rho)(\mathbf{v}_1, l') \wedge (l' = 0)] \wedge \phi \\ \Gamma_1 = \Gamma[(l, \text{unchecked}) \mapsto \Gamma(l, \text{unchecked}) \wedge (\varphi_1 \vee \varphi_2)] \\ \phi, \rho, \Gamma_1 \vdash_{\text{safety}}^i \mathbf{v}_1 := \text{access\_ok}(\mathbf{v}_2) : \Gamma_2 \end{array}}{\phi, \rho, \Gamma \vdash_{\text{untrusted}}^i \mathbf{v}_1 := \text{access\_ok}(\mathbf{v}_2) : \Gamma_2}$$

**Dereferences** The dereference of a pointer  $\mathbf{v}_2$  is `unsafe` when  $\mathbf{v}_2$  is an `unchecked`, `untrusted` pointer. Consider the statement  $\mathbf{v}_1 := * \mathbf{v}_2$  that dereferences  $\mathbf{v}_2$ . The location  $l$  is `unsafe` after the dereference when either  $l$  is `unsafe` before the dereference or  $\mathbf{v}_2$  points-to  $l$  and  $l$  is `untrusted` and  $l$  is `unchecked`. The location  $l$  is `unsafe` before the dereference under the guard  $\Gamma(l, \text{unsafe})$  where  $\Gamma$  is the state environment before the dereferencing statement. Now,  $\varphi_1 \equiv \text{rval}(\rho)(\mathbf{v}_2, l) \wedge \phi$  is the guard under which  $\mathbf{v}_2$  points-to  $l$ , and  $\varphi_2 \equiv \Gamma(l, \text{untrusted}) \wedge \Gamma(l, \text{unchecked})$  is the guard under which  $l$  is `untrusted` and

**unchecked**, where  $\rho$  is the points-to graph at the dereferencing statement and  $\phi$  is the guard under which the dereferencing statement executes. The following inference rule describes how the analysis updates the state environment with the guard under which  $l$  is **unsafe**.

$$\frac{\begin{array}{c} l \in Loc \\ \varphi_1 \equiv rval(\rho)(\mathbf{v}_2, l) \wedge \phi \\ \varphi_2 \equiv \Gamma(l, \mathbf{unsafe}) \wedge \Gamma(l, \mathbf{unchecked}) \\ \Gamma_1 = \Gamma[(l, \mathbf{unsafe}) \mapsto \Gamma(l, \mathbf{unsafe}) \vee (\varphi_1 \wedge \varphi_2)] \\ \phi, \rho, \Gamma_1 \vdash_{safety}^i \mathbf{v}_1 := * \mathbf{v}_2 : \Gamma_2 \end{array}}{\phi, \rho, \Gamma \vdash_{untrusted}^i \mathbf{v}_1 := * \mathbf{v}_2 : \Gamma_2}$$

## 6 Interprocedural Analysis

At call statements, the analysis looks up the summary state environment of the callee generated in the previous iteration, instantiates it in the calling context, and applies it at the call site:

$$\frac{\begin{array}{c} \vdash_{safety}^{i-1} \mathbf{t}_1 \ \mathbf{Q} \ (\mathbf{t}_2 \ \mathbf{v}) \ \{ \mathbf{s} \} : \Gamma_{sum}^Q \\ \Gamma_1^P = \mathcal{I}_{StateEnv}(\rho^P, \Gamma^P)(\Gamma_{sum}^Q) \\ \Gamma_2^P = refine(\Gamma_1^P, \phi^P) \end{array}}{\phi^P, \rho^P, \Gamma^P \vdash_{safety}^i \mathbf{v}_1 = \mathbf{Q} \ (\mathbf{v}_2) : \Gamma_2^P}$$

The first antecedent binds  $\Gamma_{sum}^Q$  to the summary state environment of procedure  $Q$  generated in the  $(i-1)$ th iteration. The second antecedent binds the state environment  $\Gamma_1$  to the *state environment instantiation* of  $\Gamma_{sum}^Q$ . The last antecedent binds  $\Gamma_2$  to the refinement of  $\Gamma_1$  with  $\phi$ . The remainder of this section explains how the analysis instantiates a state environment using  $\mathcal{I}_{StateEnv}$ . Intuitively, the process is similar to the matching and substitutions performed in type checking a type of a polymorphic function's domain against the type of the argument, however, this situation is more involved because of the presence of possible aliasing and guards.

As in the rule above,  $\Gamma_1^P$  is the state environment instantiation of the callee state environment  $\Gamma_{sum}^Q$ . We describe how the guard  $\Gamma_1^P(q_2^P)$  associated with a state  $q_2^P \in State^P$  is calculated given the state environment  $\Gamma$  and points-to graph  $\rho^P$  before the call statement. The definition uses the *state instantiation* function  $\mathcal{I}_{State}^P$  defined in Section 6.1 and the *guard instantiation* function  $\mathcal{I}_{Guard}^P$  defined in Section 6.2. Now,  $\Gamma_1^P(q_2^P)$  is the disjunction over the guards under which some  $Q$ -state  $q_1 \in State^Q$  instantiates to  $q_2^P$  conjoined with the guard instantiation of  $\Gamma_{sum}^Q(q_1^Q)$ . Formally, the state environment instantiation function is

$$\mathcal{I}_{StateEnv}^P(\rho^P, \Gamma^P)(\Gamma_1^Q)(q_2^P) = \bigvee_{q_1^Q \in State^Q} [\mathcal{I}_{State}^P(\rho^P)(q_1^Q, q_2^P) \wedge \mathcal{I}_{Guard}^P(\rho^P, \Gamma^P)(\Gamma_1^Q(q_1^Q))]$$

## 6.1 State Instantiation

Recall that the abstract states associated with each procedure are disjoint. Thus, the analysis uses a *state instantiation* function  $\mathcal{I}_{State}$  to map an abstract state of one procedure to the corresponding abstract state in another procedure. Suppose  $P$  calls  $Q$  at call statement  $\mathbf{s}$  where  $\rho$  is the points-to graph at  $\mathbf{s}$ . The *state instantiation* function  $\mathcal{I}_{State}^P \in PointsTo^P \rightarrow (State^Q \times State^P) \rightarrow Guard^P$  gives the guard under which a callee state  $q_1^Q \in State^Q$  instantiates to caller state  $q_2^P \in State^P$ .

Consider the unchecked, untrusted pointer dereference property. The state instantiation function  $\mathcal{I}_{State}^P$  instantiates a callee state  $(l_1^Q, t) \in State^Q$  into the caller state  $(l_2^P, t) \in State^P$  when  $l_1^Q$  instantiates to  $l_2^P$  according to the location instantiation function  $\mathcal{I}_{Loc}$  described in Section 3.

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

## 6.2 Guard Instantiation

Recall that, because of the context variable abstraction performed on the preliminary summary state environment, the only atoms that appear in the guards in the final summary state environment of a procedure are context variables. Thus, the *guard instantiation* function  $\mathcal{I}_{Guard}^P \in (PointsTo^P \times StateEnv^P) \rightarrow Guard^Q \rightarrow Guard^P$  instantiates a callee guard by instantiating each of the context variables that appear in the guard. We informally explain how the guard instantiation function works with an example.

EXAMPLE. Suppose the following: procedure  $P$  calls  $Q$ ,  $\Gamma^P$  is the state environment and  $\rho^P$  is the points-to graph at the call statement,  $\{q_{1a}, q_{1b}\} \subseteq State^Q$ ,  $\{c_{1a}, c_{1b}\} \subseteq XVar^Q$  where  $xvar(q_{1a}) = c_{1a}$  and  $xvar(q_{1b}) = c_{1b}$ , and  $\{q_{2a}, q_{2b}, q_{2c}\} \subseteq State^P$ . Also suppose the state instantiation function reports that  $\mathcal{I}_{State}^P(\rho)(q_{1a}, q_{2a}) = true$ ,  $\mathcal{I}_{State}^P(\rho)(q_{1b}, q_{2b}) = \psi$ , and  $\mathcal{I}_{State}^P(\rho)(q_{1b}, q_{2c}) = \neg\psi$ . Consequently,  $q_{1b}$  may instantiate into either  $q_{2b}$  under  $\psi$  or  $q_{2c}$  under  $\neg\psi$ .

Consider the  $Q$ -guard  $c_{1a} \vee c_{1b}$ . Its guard instantiation consists of two cases. The first case occurs when the  $Q$ -states  $q_{1a}, q_{1b}$  instantiate into the  $P$ -states  $q_{2a}, q_{2b}$  which occurs under the guard  $\mathcal{I}_{State}^P(\rho)(q_{1a}, q_{2a}) \wedge \mathcal{I}_{State}^P(\rho)(q_{1b}, q_{2b})$  which is equivalent to  $true \wedge \psi$ . The second case occurs when  $Q$ -states  $q_{1a}, q_{1b}$  instantiate into the  $P$ -states  $q_{2a}, q_{2c}$  which occurs under the guard  $\mathcal{I}_{State}^P(\rho)(q_{1a}, q_{2a}) \wedge \mathcal{I}_{State}^P(\rho)(q_{1b}, q_{2c})$  which is equivalent  $true \wedge \neg\psi$ .

To perform the guard instantiation of  $c_{1a} \vee c_{1b}$ , first recall that  $c_{1a}$  is the context variable for  $q_{1a}$  and  $c_{1b}$  is the context variable for  $q_{1b}$ . Now, we examine the first case where  $q_{1a}, q_{1b}$  instantiate into  $q_{2a}, q_{2b}$ . In this case,  $c_{1a}$  polymorphically represents the guard associated with  $q_{2a}$  in the state environment at the call site, which is  $\Gamma^P(q_{2a})$ . Similarly,  $c_{1b}$  polymorphically represents  $\Gamma(q_{2b})$ . Consequently,  $c_{1a} \vee c_{1b}$  instantiates into  $\Gamma^P(q_{2a}) \vee \Gamma^P(q_{2b})$  in the first case. The second case is similar except that, since  $q_{1b}$  instantiates into  $q_{2c}$  instead of  $q_{2b}$ , the guard  $c_{1a} \vee c_{1b}$  instantiates into  $\Gamma^P(q_{2a}) \vee \Gamma^P(q_{2c})$ . Finally, we combine the two cases by disjoining the first case instantiation refined with  $\psi$  and the second

case instantiation refined with  $\neg\psi$ . Thus, the final guard instantiation of  $c_{1a} \vee c_{1b}$  is  $([\Gamma^P(q_{2a}) \vee \Gamma^P(q_{2b})] \wedge \psi) \vee ([\Gamma^P(q_{2a}) \vee \Gamma^P(q_{2c})] \wedge \neg\psi)$ .  $\square$

## 7 Refining the Initial State Environment

Recall that analyzing a procedure using the polymorphic initial state environment generates a summary that may be used in any calling context including those that do not appear in the program (recall Section 5.1). The cost of such a pure, compositional bottom-up analysis is exactly that it must account for the possibility of every possible environment which, depending on the application, may be prohibitively expensive.

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

The function  $statecontext \in (Proc \times \mathbb{N}) \rightarrow 2^{State}$  associates each procedure  $Q$  with the set of states that appear in some calling context with a satisfiable guard in iteration  $i$ . Formally,  $q_1^Q \in statecontext(Q, i)$  if and only if the judgment

$$\phi^P, \rho^P, \Gamma^P \vdash_{safety}^i \mathbf{v}_1 = \mathbf{Q}(\mathbf{v}_2) : \Gamma'^P$$

holds in a procedure  $P$  and

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

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  $\Gamma^P(q_2^P)$  in the calling context is satisfiable. Finally, we define a new *satisfiability initial state environment*  $\iota_{sat}$  for procedure  $Q$  in the  $i$ th iteration that incorporates the satisfiability of guards in the calling contexts:

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

## 8 Evaluation

This section evaluates an implementation of the unchecked, untrusted pointer dereference analysis over the entire Linux 2.6.17.1 distribution for the x86 architecture. The distribution contains over 6.2 million lines of code and 91,543 procedures. Eliminating unchecked, untrusted pointer bugs from Linux has been a significant focus of Linux development over the last several years, and the kernel

has thousands of annotations indicating where programmers believe untrusted pointers can arise. These annotations are used, but not checked for correctness, by Sparse, a type qualifier system [8] developed by Linus Torvalds. Sparse is not sound, but the systematic effort of the combined use of Sparse and the annotations makes it quite plausible that the kernel is actually free of these errors, and indeed we found no such bugs during our experiment. Our goal is to measure how close we come to verification, i.e., producing no warnings at all. Note that our implementation does not make use of the Sparse annotations.

We wrote summaries for three macros and procedures designated by operating system developers as primitives that check whether an untrusted pointer points into user space. We also annotated several frequently used inline assembly statements with summaries of their behavior. Our implementation is built on Saturn [1]. We ran the Saturn alias analysis to compute the memory model for each procedure (recall Section 3) and then ran our safety analysis. The analysis times out on 154 procedures, or 0.17% of the procedures.

We used two additional annotations beyond the summaries mentioned above to scale the analysis to a fixed point. These annotations soundly restrict which locations are tracked as `untrusted` at specific program points. One annotation refines the guard under which a particular location in procedure `HiSax_command` is tracked as being `untrusted` with additional *interprocedural*, path-sensitive information. The other annotation occurs in `notifier_call_chain`, a generic procedure that takes a function pointer argument  $f$  and a pointer argument  $u$ , which may be untrusted depending on the value of  $f$ , and calls  $f(u)$ . Because our analysis does not track the correlation between  $f$  and  $u$  we conclude that `notifier_call_chain` passes an untrusted pointer to every possible target of  $f$ .

## 8.1 Results

An untrusted pointer *source* is a pointer argument to a system call; an untrusted 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 untrusted pointer sources (or 98.2% of sources) and 851,686 of the 852,092 untrusted pointer sinks that do not appear in procedures that time out (or 99.95% of sinks). There were 11 warnings on untrusted pointer sources (1 source warning for approximately 560,000 lines of code) and 406 warnings on untrusted pointer sinks (1 sink warning for approximately 15000 lines of code) all of which can be discharged by 22 additional, simple annotations. Almost all false positives can be classified into two categories: lack of interprocedural path sensitivity and imprecision in analyzing function pointers.

## 8.2 Interprocedural Path Insensitivity

The analysis presented in this paper is fully *intraprocedurally* path-sensitive but *interprocedurally* path-*insensitive*. 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

```

1: int sound_ioctl(..., uint cmd, ulong /*user*/ arg) {
2:     if (_SIOC_DIR(cmd) != _SIOC_NONE && _SIOC_DIR(cmd) != 0)
3:         if(_SIOC_DIR(cmd) & _SIOC_WRITE)
4:             if (!access_ok(..., arg,...))
5:                 return -EFAULT;
6:     ...
7:     return sound_mixer_ioctl(..., cmd, arg);
8: }
9: int sound_mixer_ioctl(uint cmd, void /*user*/ *arg) {
10:    ...
11:    return aci_mixer_ioctl(...,cmd, arg);
12: }
13: int aci_mixer_ioctl(..., uint cmd, void /*user*/ *arg) {
14:     switch(cmd)
15:         case SOUND_MIXER_WRITE_IGAIN:
16:             ...*arg...;
17:     ...
18: }

```

**Fig. 2.** Procedures in `sound/oss/soundcard.c` from Linux 2.6.17.1

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 untrusted pointer sources and 265 untrusted pointer sinks.

Consider the procedure `sound_ioctl` of `sound/oss/soundcard.c` from Linux 2.6.17.1 (see Figure 2) where the formal parameter `arg` is an untrusted pointer passed from the system call `sys_ioctl`. Line 4 performs a check on the untrusted pointer using the special checking primitive `access_ok` under a condition  $\pi(\text{cmd})$  based on special bits of `cmd` selected by the macros `_SIOC_DIR`, `_SIOC_NONE`, and `_SIOC_WRITE`. Thus, before the call to `sound_mixer_ioctl` on line 7, `arg` is checked under the condition  $\pi(\text{cmd})$ . Consequently, any subsequent dereference of `arg` must be guarded by a condition that implies  $\pi(\text{cmd})$ . Line 16 in procedure `aci_mixer_ioctl` dereferences the untrusted pointer `arg` under the condition `cmd == SOUND_MIXER_WRITE_IGAIN` which implies  $\pi(\text{cmd})$ , and thus the untrusted 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.

### 8.3 Function Pointers

Four untrusted pointer sources and 130 untrusted 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 3. This single site is responsible for the analysis failing to discharge 1 untrusted pointer source and

```

1: struct { char *name; ...} map[] = ...,
2:     {NFSCCTL_GETFD} = {.name = ".getfd", ...},
3:     {NFSCCTL_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: }

```

**Fig. 3.** System call in fs/nfsctl.c from Linux 2.6.17.1

111 untrusted 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` 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`.

## 9 Related Work

We briefly describe closely related work on scalable verification of finite-state properties. CQual [8] is a type-qualifier inference system that has been used to try to verify the absence of unchecked, untrusted pointer dereferences in earlier versions of Linux [12]. While CQual could be summary-based, this particular implementation represents the entire program in main memory, limiting 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 a warning for every 1300 lines of code. Our order of magnitude improvement in warning density is due to the fact that our system is flow-sensitive and intraprocedurally path-sensitive, features that CQual currently lacks. Several sources of false positives listed in [12] do not appear in our experiments, because our analysis is precise enough to handle them soundly without warnings. Interestingly, function pointers are not seen as a major cause of false positives in [12]; this is actually consistent with our experience, which is that accuracy in analyzing function pointers does not have much effect until one analyzes the full kernel, with all device drivers linked in.

ESP [5] is a path-sensitive dataflow analysis system used to verify *probing*, the Windows version of unchecked, untrusted pointer dereferences [6]. Their experiment reports 50 warnings on untrusted pointer sources in a much smaller body of code (about one million lines), all of which were false positives (as with Linux, there has been a concerted effort to remove such bugs from Windows in recent years). 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 Saturn's than CQual's. One significant difference is that ESP encodes path sensitivity by tracking sets of dataflow facts, one set for each path. In contrast, we associate a formula (the guard) with each fact describing a set of paths. In the Saturn representation it is easy to reason about multiple paths simultaneously, as the guards directly encode all paths where a fact holds.

Sparse (recall Section 8) is a type qualifier checker for validating untrusted pointer dereferences. The checker requires an annotation on every pointer variable declaration that could contain an untrusted pointer. Linux 2.6.17.1 has over 9000 such annotations, representing a great deal of manual effort. Sparse reports hundreds of warnings, and Sparse is also not completely sound. Our system is sound and produces fewer warnings with a handful of annotations.

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.

## 10 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, untrusted pointer dereferences in the Linux operating system.

## 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. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.
5. 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.
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. 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.