Path-Sensitive Dataflow Analysis with Iterative Refinement

Dinakar Dhurjati[1], Manuvir Das[2], and Yue Yang[2]

[1] University of Illinois at Urbana Champaign
{dhurjati@cs.uiuc.edu}
[2] Center for Software Excellence, Microsoft Corporation
{manuvir, jasony@microsoft.com}

Abstract. In this paper, we present a new method for supporting abstraction refinement in path-sensitive dataflow analysis. We show how an adjustable merge criterion can be used as an interface to control the degree of abstraction. In particular, we partition the merge criterion with two sets of predicates — one related to the dataflow facts being propagated and the other related to path feasibility. These tracked predicates are then used to guide merge operations and path feasibility analysis, so that expensive computations are performed only at the right places. Refinement amounts to lazily growing the path predicate set to recover lost precision. We have implemented our refinement technique in ESP, a software validation tool for C/C++ programs. We apply ESP to validate a future version of Windows against critical security properties. Our experience suggests that applying iterative refinement to path-sensitive dataflow analysis is both effective in cutting down spurious errors and scalable enough for solving real world problems.

1 Introduction

In recent years, model checking and dataflow analysis have emerged as competing approaches for compile-time defect detection. Many model checking tools [1,2,3,4,5,6] have enjoyed the benefit of abstraction refinement. The process starts with a coarse (simple) abstract model. Once the model is checked, the feasibility of the resulting abstract counterexample is examined. If feasible, the result reveals a real error; otherwise, the result is a false positive. In the latter case, the abstraction is refined and fed back to the model checker for successive checking.

Although abstraction refinement has proven effective in model checking, similar techniques have not been widely adopted in dataflow analysis [7,8,9,10]. The main difficulties involved in supporting abstraction refinement in dataflow analysis are: (1) How do we control the abstraction? Model checking has an a priori abstract model, which naturally represents the abstraction; in dataflow analysis, the abstraction is computed directly from a source program as dataflow facts by obtaining the least fixed point solution to a set of transfer functions. (2) How do we identify false positives? The abstract counterexample produced by a model checker is a single path. Therefore, the model checking refinement process can

K. Yi (Ed.): SAS 2006, LNCS 4134, pp. 425–442, 2006.

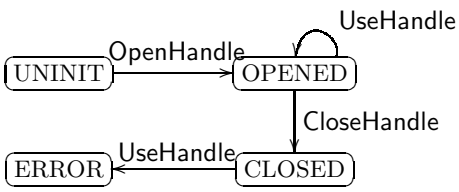
prove its feasibility using theorem proving. In contrast, the abstract trace generated by dataflow analysis is a summary of up to exponentially many (or even infinitely many) execution paths in the original program — rigorously proving its feasibility is very difficult.

In this paper, we address these difficulties. Our first insight is that the merge criterion in dataflow analysis plays a key role in determining path-sensitivity. By extending the *fixed* merge criterion to an *adjustable* one, abstractions can be tuned in response to the characteristics of the paths being analyzed. Our second insight is that in the absence of a single counterexample, the refinement process can be effective if spurious error traces, and the missing predicates that lead to these traces, can be identified heuristically. By combining an adjustable merge criterion with efficient heuristics for finding where precision is lost, the analysis can adaptively adjust its precision in a demand-driven fashion. We apply these refinement techniques to ESP [10,11], a validation tool for large C/C++ programs, with promising results.

Loss of precision in dataflow analysis mainly arises in two situations: (1) when states along different paths are merged at merge points and (2) when dataflow facts are propagated across infeasible paths. Consider ESP as an example, its path-sensitive dataflow analysis incorporates two mechanisms to preserve the precision of path-sensitive analysis: an effective merge algorithm based on *property simulation* [10] and a path feasibility analyzer called SSM (Symbolic State Manager) [12]. However, these mechanisms are "static", in the sense that they are based on a fixed set of rules. Although ESP can preserve precision in common cases, sometimes it is still overly conservative — the merge may be too aggressive or the tracking on simulation states may be too imprecise. As a fixed-precision analysis, ESP is not able to recover precision loss should it arise. The iterative refinement technique described in this paper removes this limitation.

Example. To illustrate how a false positive may be introduced in static analysis, consider the example in Figure 1. The finite state machine in Figure 1(a) encodes a property adapted from the requirement on Windows kernel objects. We intend to validate that a program should not call `UseHandle` on an object that has been previously closed by `CloseHandle`. Figure 1(b) shows a function that processes a handle according to various status values. Albeit simple, this code snippet reflects a common coding practice: Programmers often use internal flags (such as $flag1$ and $flag2$ in this example) to record programming states and later use them to guide control flow. If an analysis does not track enough branch correlations, it may report a violation to the property protocol at `L4`. In contrast, a more precise analysis reveals that no valid error path can lead to this point because of the guard on $flag1$ at `L3`. Hence, the warning is a false positive. □

Contributions. A practical analysis tool is aimed at supporting industrial programs with complex language features. Therefore, it must satisfy several challenging (and sometimes conflicting) goals at once: (1) it should scale to large programs, (2) it should offer enough precision, and (3) it should produce use-



(a)

```
        extern int stat1, stat2;
        void Process(int handle) {
            int flag1, flag2;
L1:         if (stat1 > 0) {
                CloseHandle(handle);
                flag1 = 1;
            }
            else
                flag1 = 0;

L2:         if (stat2 > 0)
                flag2 = 1;
            else
                flag2 = 0;

L3:         if (flag1 == 0)
L4:             UseHandle(handle);
        }
```

(b)

Fig. 1. Usage of Windows kernel objects. Figure (a) shows the finite state machine of the typestate property. Figure (b) shows a simplified code pattern adapted from the Windows kernel.

ful feedback to help programmers investigate bugs. This paper addresses these tradeoffs and integrates practical techniques to produce useful results. Our contributions can be summarized as follows.

- We show how the merge criterion in dataflow analysis can be extended to support *adaptable abstractions*. In particular, we apply two predicate sets to control precision: one is a fixed set consisting of predicates related to the property of interest, and the other is an adjustable set consisting of predicates related to path feasibility (these predicates are referred to as *path predicates*).
- We apply the following heuristic to quickly recognize potential false positives: Along with an error trace, there often exists a corresponding *good trace* (a path leading to a valid property state) that ends at the same program point. If these "good state" and "error state" co-exist, it is likely that the error state is reached because useful predicates present in the good state have been inadvertently dropped from the error state during the analysis. Therefore, addition of these distinguishing predicates to the tracked path predicate set will help eliminate spurious errors.
- We implement the refinement techniques in ESP. We show experimental evidence that iterative refinement is effective in filtering out false positives and is scalable enough to be applied to large-scale programs.

The remainder of the paper is organized as follows. In Section 2, we review the techniques applied in ESP. In Section 3, we describe our iterative refinement

approach. In Section 4, we discuss experimental results. In Section 5, we review related work. We conclude in Section 6.

2 Background

ESP checks program properties related to *typestates* [13]: For a value created during program execution, its ordinary type is invariant but its typestate may be updated by certain operations. ESP allows a user to write a custom specification encoded in a finite state machine, as illustrated in Figure 1(a), to describe typestate transitions. According to the specification, ESP instruments the source program with the state-changing events. It then employs an inter-procedural dataflow analysis algorithm, based on *function summaries* [8], to compute the typestate behavior at every program point. To obtain path-sensitivity, ESP uses the combination of property simulation [10] and path simulation [12].

2.1 Property Simulation

In ESP, a symbolic state is divided into the *property state* (typestate according to the specified protocol) and the *simulation state* (state related to path feasibility). The property simulation algorithm defines a merge heuristic centered around the property of interest: At a merge point, if two symbolic states have the same property state, ESP merges the simulation states. Otherwise, ESP explores the two paths independently as in a full path-sensitive analysis.

The simulation state is an element in a lattice whose elements are abstractions of heap and store. Let D be the domain of property states and S be the domain of simulation states. Given a symbolic state $s \in S$, we denote its property state by $ps(s)$ and its simulation state by $ss(s)$. The merge criterion for s is characterized by the following grouping function:

$$\alpha_{ps}(s) = \{[\{d\}, \sqcup_{s' \in s[d]} ss(s')] \mid d \in D \wedge s[d] \neq \phi\}$$

where $s[d] = \{s' | s' \in S \ \wedge \ d = ps(s')\}$ and $\sqcup$ is the least upper bound in the simulation state lattice.

Example. To see the different effects caused by various merge policies, consider Figure 1(b). Assume that the property state is `OPENED` when function `Process` is entered. A full path-sensitive analysis tracks all paths reaching program point `L3`, resulting in four symbolic states at `L3`:

$[\{\texttt{CLOSED}\}, \{stat1 > 0, flag1 = 1, stat2 > 0, flag2 = 1\}]$
$[\{\texttt{CLOSED}\}, \{stat1 > 0, flag1 = 1, stat2 <= 0, flag2 = 0\}]$
$[\{\texttt{OPENED}\}, \{stat1 <= 0, flag1 = 0, stat2 > 0, flag2 = 1\}]$
$[\{\texttt{OPENED}\}, \{stat1 <= 0, flag1 = 0, stat2 <= 0, flag2 = 0\}]$

ESP, on the other hand, drops the correlation between variables `stat2` and `flag2` because the branch at `L2` does not affect the property state. As a result, only

two symbolic states are kept at `L3`:

$[\{\texttt{CLOSED}\}, \{stat1 > 0, flag1 = 1\}]$
$[\{\texttt{OPENED}\}, \{stat1 <= 0, flag1 = 0\}]$

Since the dropped facts are irrelevant to the property of interest, ESP still maintains enough information to conclude that `L4` is not reachable if the property state is `CLOSED`. □

Property simulation matches the coding practice of a careful programmer: The correlation between a given property state and the program state is usually guarded in the code by branch conditions. ESP makes such implicit correlation explicit. The adjustable merge criterion developed in this paper builds upon this insight by taking additional path predicates into account as well.

2.2 Path Simulation

The symbolic path simulator, referred to as *Simulation State Manager* (SSM) in this paper, manages simulation states and acts as a theorem prover to answer queries about path feasibility. A simulation state mainly consists of two sets of information: (1) the symbolic store (mapping from locations to values) and (2) a set of constraints (or path predicates) imposed on values. These path predicates are implicitly conjuncted. To reason about facts related to path feasibility, SSM applies a decision procedure based on a set of inference rules. The path simulator performs a set of transfer functions on behalf of ESP for instructions such as assignments, branches, procedure calls (into-binding), call returns (back-binding), and merges. These transfer functions update simulation states accordingly and filter out infeasible paths.

Path feasibility analysis is undecidable in general. To guarantee convergence and efficiency, the Simulation State Manager makes conservative assumptions when necessary. While such over approximation is sound (*i.e.*, it will not miss errors), it may introduce imprecision. The refinement technique in this paper allows the analysis to start with a light-weight decision procedure and fine-tune it based on counterexamples.

2.3 Imprecision Due to Property Simulation

Although the ESP merge heuristic is precise in most cases, sometimes it can be too conservative.

Example. In Figure 2(a). The branch at `L1` does not change the property state. Therefore, ESP merges the simulation states at `L2` and loses the correlation between `stat` and `flag`. As a result, a false positive is reported at `L4`. □

Our refinement technique would pick up an additional path predicate, say `stat` > `0`, and add it to the merge criterion. This would direct the analysis to track the

```
    extern int stat;
    void Process(int handle) {
        int flag = 0;
L1:     if (stat > 0)
            flag = 1;
        else
            flag = 2;

L2:     if (stat > 0)
            CloseHandle();

L3:     if (flag != 1)
L4:         UseHandle();
    }
```

(a)

```
    extern int stat;
    void Process(int handle) {
       int flag = 0;
       if ((stat & 0x81) != 0)
           return;

       CloseHandle(handle);
L1:    flag = stat & 0x1;

L2:    if (flag != 0) {
L3:        UseHandle(handle);
       }
    }
```

(b)

Fig. 2. Examples that illustrate the need of refinement. Figure (a) shows a false positive caused by the ESP merge heuristic. Figure (b) shows a false positive caused by the path feasibility analysis.

branch at `L1` accurately since the branch arms impose different facts about the predicate. With this additional precision, the analysis can rule out the false error.

Even when an error reported by ESP is a real error, it may still be beneficial to apply refinement to "concretize" the abstract counterexample, *i.e.*, to expand the merged paths at certain branch points so that the trace can be more explicit and meaningful for inspection purposes.

2.4 Imprecision Due to Path Simulation

The Simulation State Manager uses a set of inference rules to implement the underlying decision procedure. By default, it supports a subset of congruence closure and uninterpreted functions. While it is possible to apply a heavy-weight theorem prover that combines many theories, it would significantly hinder the scalability of our analysis. Therefore, when the complexity of certain branch correlations get too complicated, the correlations would be dropped due to the lack of reasoning power in the theorem prover.

Example. In Figure 2(b), the fact `(stat & 0x81)` = `0` should hold at `L1` because the function would have returned otherwise. This fact should imply `flag` = `0` at `L2`. However, if the theorem prover does not employ inference rules to track bit-wise operations, it might not be able to deduce such information — this would result in a false positive at `L3`. □

The above example reflects a coding pattern where operations are controlled by certain bits in a flag. While tracking this is critical for certain code bases and properties, it is not important in general. Therefore, it is beneficial to start an analysis with a light-weight theorem prover and only add precision as needed.

3 Refining Dataflow Analysis

There are three key steps in the dataflow refinement process: (1) identifying "suspicious" error traces that need refinement, (2) selecting a minimal set of dropped branch correlations that may contribute to the precision loss and adding them to the merge criterion, and (3) enforcing the extended merge criterion in the subsequent iteration.

3.1 Identifying False Positives

As previously mentioned, generating one concrete counterexample out of an abstract ESP counterexample may not be scalable. The novelty of our approach is that instead of trying to ascertain that an error trace reported by ESP is a false positive before starting the refinement process, we develop an inexpensive heuristic that can identify false positives with high probability.

Definition. *Corresponding Good State* — Given a candidate error state E, its corresponding good state is defined as a symbolic state G at the same program point as E, such that the property state of G is not `ERROR`. □

Heuristic 1. Given a candidate error state E, if there exists a corresponding good state E', E is subject to refinement; otherwise, E is not subject to refinement. □

Example. At program point `L4` in Figure 2(a), the error is indicated by state [`CLOSED`, $\{stat > 0, flag \neq 1\}$]. There also exists a good state, [`OPENED`, $\{stat <= 0, flag \neq 1\}$], which correctly keeps track of the correlations. According to heuristic 1, the error at `L4` is subject to refinement. □

This heuristic is based on the following intuition: If at a program point, there only exists an error trace without any good traces, there is no evidence that the program can behave correctly. Hence, the error is likely to be real.

We now formally examine heuristic 1 using a case analysis. For a given candidate error state, there may or may not exist a corresponding good state at the same program point. Among the four possible combinations (as listed below), heuristic 1 directly supports category (1) and (4). For category (2) and (3), while the heuristic cannot properly distinguish whether the error is a false positive, the refinement policy imposed by the heuristic can still be beneficial. This explains why our inexpensive heuristic can be surprisingly effective in practice.

Category 1 [**False error, With good state**]**.** Heuristic 1 directly targets this category. A false positive in this group would be properly identified.

Category 2 [**Real error, With good state**]**.** These error states would also be profiled as plausible errors needing refinement. Although these errors are not false positives, selecting them for refinement can indeed be helpful for inspection

purposes because expanding the abstract trace will make the error message more explicit and meaningful.

Category 3 [**False error, Without good state**]**.** It may appear that our heuristic is not suitable for this category because these false errors would not be subject to refinement. This, however, is not a deficiency of the heuristic. This is because ESP is conservative — it reports all (and potentially more) paths that can reach a program point. Therefore, if the error path is infeasible and there is no corresponding good path, it means that there is no feasible path at all that can reach this program point, *i.e.*, the "false error" is part of "dead code". From a software engineering point of view, it may be desirable to reveal these as real errors.

Category 4 [**Real error, Without good state**]**.** Heuristic 1 is correct in not identifying these real errors as false positives.

3.2 Selecting New Predicates

After a potential false positive is identified, we need to determine what are the additional predicates that are most likely to improve the precision. One approach is to collect all the path predicates accumulated in the error state. However, most of these predicates are irrelevant and will result in unnecessary overhead in the subsequent analysis. To pinpoint exactly where precision is lost, we develop heuristic 2 based on the insight from heuristic 1 to gather only the relevant predicates in common scenarios.

Heuristic 2. We compare the simulation states between the candidate error state and its corresponding good state. The difference in path predicates suggests why the error path and the good path deviate. Therefore, we select these distinguishing predicates and add them to the merge criterion for the subsequent iteration. □

Example. In Figure 2(a), if we compare the predicates between the simulation state of the error state, [`CLOSED`, $\{stat > 0, flag \neq 1\}$], and the simulation state of the good state, [`OPENED`, $\{stat <= 0, flag \neq 1\}$], it is clear that $\{stat > 0\}$ is a distinguishing factor. It should therefore be added to the merge criterion. □

This method is efficient because it queries the state information that is already available as dataflow facts computed from the previous iteration; it is also effective because it allows path-sensitivity to be incrementally added at exactly the right place.

3.3 Adjusting the Merge Criterion

After additional path predicates are selected, they are added to the merge criterion. At merge points in the subsequent run, incoming states that differ with

respect to these selected path predicates will be tracked independently. This section formally describes this process.

Let $P_1, P_2, ..., P_n$ be the set of predicates that need to be added to the merge criterion in any iteration. Let T be a set of tri-values $\{1, 0, *\}$, with the elements denoting `true`, `false` and `don't know`, respectively. To track the predicates $P_1, P_2, ..., P_n$ accurately in the new run, we change the property state component to $D \times T... \times T$ (or $D \times T^n$). For a symbolic state in the new run, $[\{d, t_1, t_2, ..., t_n\}, ss]$, t_i denotes whether predicate P_i is provable from the simulation state ss using a theorem prover. If $\vdash$ denotes provability, t_i can be defined as follows:

$$\begin{aligned} t_i &= 1 \text{ if } ss \vdash P_i \\ &= 0 \text{ if } ss \vdash \neg P_i \\ &= * \text{ otherwise} \end{aligned}$$

The merge criterion is extended with a tri-value vector denoting the status of each of the predicates that we have decided to track accurately. At merge points, the incoming symbolic states are compared. If there exists a t_i such that its value is different in the two incoming states, the paths are tracked separately; otherwise, the paths are merged. Formally, the grouping function for the merge criterion is modified as follows:

$$\alpha_{ps}(s) = \{[\{d\}, \sqcup_{s' \in s[d]} s(s')] |\ d \in D \times T^n \wedge s[d] \neq \phi\ \}$$

where $s[d] = \{s' | s' \in ss \wedge d = ps(s')\}$ and $\sqcup$ is the least upper bound in the execution state lattice.

The tri-values are set to `*` in the initial state for a given run of the dataflow analysis. To obtain the next set of symbolic states after a program statement, we first invoke the same transfer functions as in the original ESP. For each predicate P_i, we then invoke the Simulation State Manager to prove P_i or $\neg P_i$. Depending on the provability, we assign a tri-value to the $(i+1)^{th}$ location in the property state component. Once we have the new set of symbolic states we use the grouping function, just like in the original ESP algorithm, to merge the symbolic states whose new property states (the original property state and the predicate vector) are the same.

Example: In Figure 2(a), a subsequent run adds the predicate $\{stat > 0\}$ to the property state component. Let p denote the predicate $\{stat > 0\}$. The new ESP run at program point `L2` would have two separate symbolic states [{`CLOSED`, $p = 1$}, $\{stat > 0, flag = 1\}$] and [{`CLOSED`, $p = 0$}, $\{stat <= 0, flag = 2\}$]. Since the the property state components are no longer the same, the two symbolic states will be tracked independently. With this additional correlation, the new run will not produce an error state. □

New Automata of State-Changing Events. Changing the merge criterion as outlined above could be viewed as using an alternative property state ma-

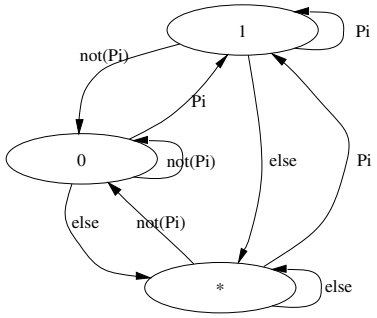


Fig. 3. A tri-value automaton for the ith predicate P_i. Values `1`, `0`, and `*` denote `true`, `false` and `don't know`, respectively.

chine for defining state-changing events. This new state machine is the product automaton of the original property automaton with n tri-value automata. The i^{th} tri-value automaton is shown in Figure 3. The product property automaton has $|D| * 3^n$ states, where $|D|$ is the number of states in the original property automaton.

Complexity of the New Merge Algorithm. The original ESP algorithm is polynomial in $|D|$ [10]. The complexity of an ESP run with refined abstraction as described above is exponential in the number of predicates since the size of our new property automaton is exponential in the number of predicates. In practice, however, this is not an issue since only a few predicates matter at any point in the program and most states in the new property state automaton are never reached. Note that we also need to add the cost of $2 * n$ calls to the theorem prover in addition to the cost of the transfer function at each step in the ESP algorithm. We can optimize away most of these queries by using a simple value flow analysis [14] to "slice" the program.

3.4 A More Refined Abstraction Method

The tri-vectors succinctly encode a set of abstract states of the program. However, this is only an approximation (referred to as the *cartesian approximation* in SLAM [15]) to the most precise predicate abstraction. For example, suppose we have added a new predicate $x = 0$. If at the merge points the facts along the two paths are $x < 0$ and $x > 0$ respectively, sometimes it is desirable to keep the fact of $x \neq 0$ after the merge. However, with cartesian approximation, such information is lost.

To regain the precision lost in this situation, we have also implemented an alternative abstraction method based on *distinguishing variables*. Instead of using the tri-value of a predicate to control merging, we project the related variables from the predicate set. At merge points, any incoming states with different facts on the selected variables are tracked separately. In the above example, x would

```
procedure IterativeRefine(P, D)
begin
  for each f ∈ P
    E[f] = φ;
  while true do
    switch ESPRun(P, D, E)
    case SUCCESS:
      output "success"; break;
    case FAILURE(T): //T is a set of ESP error traces
      for each f ∈ P
        E'[f] = φ
      for each t ∈ T
        if (CheckSuspiciousTrace(t)) then
          E' = E' ∪p RelevantPredicates(t)
        if (E' ⊆ E) then
          output T as error; break;
    E = E ∪p E';
end
```

Fig. 4. The iterative refinement algorithm for checking if program P satisfies the property specified in automaton D

be picked out as a controlling factor for merge. Consequently, the incoming paths with $x < 0$ and $x > 0$ are not merged since they affect the value of x.

Essentially this approach enables a mechanism for lazily tracking disjunctions on predicates. Initially, the constraints carried in a simulation state are implicitly conjuncted since it is prohibitive for the Simulation State Manager to eagerly track every disjunction on merge due to the exponential cost. With the variable-based refinement technique, an application is allowed to, and can afford to, have a "deep" analysis when necessary, because it only focuses on a small number of selected variables. Instead of asking the Simulation State Manager to explicitly track the disjunctions, the merge mechanism is used to split the paths along which the disjunctions are tracked independently.

One caveat with this refined abstraction method is that the fixed point computation might not converge when the tracked variable is updated inside a loop. To guarantee termination we unroll such loops for a fixed number of iterations.

3.5 The Iterative Refinement Algorithm

Having discussed the key steps in the refinement process, we now present the iterative refinement algorithm for ESP.

Our analysis starts with the original ESP merge criterion and a set of simple inference rules for reasoning about path feasibility. It then repeats the following process: It first uses heuristic 1 to identify suspicious error traces. It then uses heuristic 2 to collect the missing path predicates. Based on these new predicates, it constructs a new precision policy by adding those selected predicates (or projected variables) to the merge criterion. Finally, it conducts a more thorough analysis

along a focused set of paths. During the new iteration, whenever the tracked predicates are involved, paths will be kept separate at merge points and comprehensive inference rules in Simulation State Manager will be enabled in path simulation.

The iterative refinement algorithm for ESP is shown in Figure 4. Given a program P and a property state automaton D, procedure **IterativeRefine** tries to verify if P satisfies D by iteratively calling **ESPRun**. Procedure **ESPRun** takes P, D, and E as input, where E is a map between functions and sets of predicates: For a function f, E[f] represents the predicates that should be tracked accurately when analyzing function f. **ESPRun** performs an analysis similar to the original ESP, except that it uses the new precision policy based on E[f]. If **ESPRun** returns success, *i.e.*, it reports no error traces, P satisfies the property specified in D. Otherwise, **ESPRun** returns a set of error traces T. We then apply heuristic 1 to identify suspicious error traces in procedure **CheckSuspiciousTrace**. If we find a suspicious error trace, we use heuristic 2 to generate a set of relevant predicates from the error trace in procedure **RelevantPredicates**. $\cup_p$ can be succinctly described using the following equation: for each f, (E $\cup_p$ E')[f] = E[f] $\cup$ E'[f]. Essentially it aggregates the predicates for each function. If there are no new predicates that need to be tracked accurately, we stop and report the remaining errors as real errors. Otherwise we do one more iteration by accurately tracking the predicates in E $\cup_p$ E'.

3.6 Incremental Refinement

A more precise analysis applied in a later iteration will explore only a subset of the paths explored by the analysis in previous iterations. In particular, this means that a symbolic state that does not reach the error state in a previous iteration is guaranteed not to reach the error state in subsequent iterations.

We have implemented a form of this optimization in ESP, at all call sites. In a given iteration, we record all of the property states that lead to an error state. In the following iteration, we terminate the analysis for symbolic states at call sites whose property state did not lead to an error state in the previous iteration. This optimization allows us to avoid repeating part of the computation from previous iterations.

4 Results

With the integration of refinement techniques, ESP has been successfully deployed in validating several critical security properties for a future version of Windows. In this section, we summarize our general experience on root cause of false positives and use one of the properties as a case study.

4.1 General Experience

We have studied the root cause of false positives by manually inspecting the error traces from various experiments using the Windows code base. We categorized false positives in several groups: (1) those introduced by the imprecision of our

analysis, (2) those introduced by the limitation of our specification method, and (3) those that are "real errors" according to the specification, but the violation to the protocol is by design, e.g., for performance reasons.

The distribution of these categories varies depending on properties being checked, but the first group is usually the most common case. Among this group, we have found that in most cases, the false positives are introduced by lacking reasoning power from the Simulation State Manager. This further confirmed that the ESP merge heuristic is usually precise enough.

The second group of false positives is due to "under specified" properties. This issue arises in practice when specifications are developed based on coarse documentations and program flags are used to track more state transitions than the transitions specified in the property automaton. When these flags are assigned with different values that appear to be "irrelevant" with respect to the specification, the correlations would be lost due to excessive merging. Our refinement method allows ESP to identify the additional branch correlations and track those accurately without requiring refinement of the specifications.

Traces in the third group are "benign". However, they are still worth a careful code review because other invariants are usually required to maintain program correctness.

4.2 Case Study

We have performed a case study by checking a security vulnerability. This vulnerability arises when a program acquires exclusive access to a system resource and then relinquishes the exclusive access by closing its handle but "leaks" the access to the resource through certain API. Checking this kind of property has been difficult with previous tools because it requires precise tracking of value flow in large programs[1].

Scalability: Using two PCs, each with a 3.06GHz Xeon CPU and 2GB of RAM, iterative refinement completed in 765 minutes for ASTs covering 5079 binaries (DLL, SYS, and EXE modules) in a future version of Windows. ESP discovered 83 traces, out of which 47 were confirmed as real bugs and fixed. Being able to perform such an analysis for the whole Windows code base clearly demonstrates the scalability of our approach.

Effectiveness of Heuristic: After those 83 traces were produced, we applied our heuristic again to further classify which of them are likely false positives. We partitioned the traces into two buckets. If at a program point, there only exists an error trace without any good traces, we put it into the high-confidence bucket. If the trace has a corresponding good path, we put it into the low-confidence bucket. The high-confidence bucket contains 38 real bugs and 2 spurious warnings, with a noise ratio of 5%. The low-confidence bucket, on the other hand, contains 9 real bugs and 34 spurious warnings, with a noise ratio of 79%. This

[1] We are unable to provide more details about violations of the property because some instances identified by ESP apply to previously shipped products as well.

ranking method allows us to quickly provide the high-confidence bucket to developers and focus on the low-confidence bucket to figure out where the tool needs to be improved. The final spurious traces are mostly due to complex code patterns that are too hard to track. Our existing Simulation State Manager is not powerful enough to handle these cases. Nonetheless, our heuristic has been shown to be highly effective. In our future work, we plan to power up the Simulation State Manager so that the remaining false positives can be further reduced with iterative refinement.

Precision improvement: We also conducted a comparison analysis between using and not using the refinement technique, which has shown that the refinement technique is effective in precision enhancement. As an example, we made two runs on an EXE binary with 11718 LOC and 388 functions, one with refinement and the other without refinement. When refinement is applied, ESP reports 1 real bug and no false positives in 6.9 seconds. When refinement is not used, ESP also reports 3 spurious errors along with the actual bug in 3.2 seconds.

To summarize, these experiments suggested that (1) it is feasible to integrate abstraction refinement to diagnose industrial-sized programs, (2) the heuristic for distinguishing potential false positives is effective, and (3) refinement can improve precision with a modest performance cost.

5 Related Work

The main contribution of this paper is an approach that integrates abstraction refinement, inter-procedural dataflow analysis, and counterexample-based heuristics in a novel way to provide a practical solution for improving software quality. Since our analysis draws on several insights from previous work, there are several categories of related work.

5.1 Path Feasibility Analysis

There is a long line of work on improving path feasibility in dataflow analysis. For example, qualified dataflow analysis [16] uses a given set of assertions on variable values to "qualify" paths under consideration. Bodik et al. [9] mark infeasible paths to improve the accuracy of def-use pair analysis. ESP [10,11], the basis of this paper, uses specification states to distinguish merge policy at merge points and relies on symbolic path simulation to enforce path feasibility. These works all use a pre-defined set of qualifications and do not address refinement issues.

5.2 Demand-Driven Analysis

Our approach is similar to demand-driven analysis [8,17] but addresses different issues. Their algorithms delay the computation of part of the analysis until it is needed; but the analysis is performed with *fixed* precision. Our analysis uses function summaries described in [8] for inter-procedural analysis. In addition, our algorithm delays adding precision to the analysis until it is dictated by evidence.

5.3 Iterative Refinement

Several dataflow algorithms [18,19,20,21] have applied refinement techniques for iteratively adjusting precision. While conceptually similar to our refinement process, these algorithms are domain-specific. For example, Guyer et al. [18] use client-driven refinement for pointer analysis and Plevyak et al. [19] use refinement for inferring concrete types in objet-oriented languages. Trace partitioning discussed in [20,21] focus on deciding which explicit disjunctions to keep during the analysis. In contrast to these analyses, our refinement technique targets more flexible typestate defect detection and can be applied to regain lost facts.

Fischer et al. [22] describe *predicated lattices*, a technique that is close in spirit to our refinement approach. Their framework partitions the program state according to a set of predicates and tracks a lattice element for each partition. Our abstraction mechanism based on the adjustable merge criterion could be viewed as a practical way of implementing predicated lattices. Their work focuses on a general framework and does not address how to pick additional predicates for refinement. In contrast, our particular interest is in combining effective partitioning of predicates with efficient heuristics to recognize important predicates.

5.4 Abstraction Refinement in Model Checking

Abstraction refinement has been an accepted technique in model checking tools, e.g., [1,2,3,4,6,23]. Our approach is different in how program abstraction is represented, selected, and enforced. (1) In model checking, the abstraction is defined by the state space of a given abstract model. In our analysis, the abstraction is done on the fly via selective merging. (2) Model checking techniques use theorem proving to map an abstract counterexample to a concrete program trace to identify false positives. While generating new predicates from the proof of unsatisfiability is more accurate and complete, it is also more expensive. We use an inexpensive but effective heuristic to identify suspicious counterexamples and then check the feasibility during the extra iteration of ESP analysis. (3) Model checking starts with a coarse abstraction and requires an expensive iteration process to reach the ideal abstraction. In contrast, ESP reports very few false positives to begin with. Hence, our seed abstraction is much closer to the desired one, which leads to faster convergence. Studies in [22,24] also show that it is beneficial to use ESP to provide a starting abstraction for the refinement process.

5.5 Counterexample-Based Heuristics

Counterexample-based heuristics are developed in [25,26]. Our heuristic for finding missing predicates is inspired by these works. The main difference is that these works focus on explaining root causes of real errors — they try to pinpoint deviating path segments between error paths and good paths. In contrast, we use the existence of corresponding good traces to quickly identify potential false positives and useful predicates.

5.6 Precise Symbolic Simulation

Several static tools, such as PREfix [27], CMC [28], and Saturn [29], provide precise symbolic simulation but limit exploration of program paths. We offer a complementary approach: instead of truncating the search space, our analysis considers all paths but guides the exploration effort to where it is most productive.

6 Conclusions

A static tool must walk the fine line between precision and scalability. In this paper, we have presented a new approach that allows the dataflow abstraction to be refined incrementally in response to the characteristics of the paths being analyzed. We have used the refinement technique to help validate a future version of the Windows operating system against important security properties. Our experience suggests that the heuristic for finding false positives is highly effective and the refinement method is scalable enough to be of practical use.

Acknowledgments. We thank Stephen Adams, Zhe Yang, Vikram Dhaneshwar, and Hari Hampapuram from the Center for Software Excellence at Microsoft for their infrastructure support and insightful suggestions. Stephen Adams also conducted some experiments. We are also grateful to Tom Ball from Microsoft Research for many valuable discussions.

References

1. Thomas Ball and Sriram Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2002.
2. Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
3. Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.
4. T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2004.
5. Cormac Flanagan. Automatic software model checking using CLP. In *ESOP*, pages 189–203, 2003.
6. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
7. 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 ACM Symposium on Principles of Programming Languages (POPL)*, 1977.

8. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural data flow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1995.
9. Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *Proceedings of the Sixth European Software Engineering Conference*, 1997.
10. Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, 2002.
11. Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004.
12. Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulatin in path-sensitive datflow analysis. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2005.
13. R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.
14. Manuvir Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, 2000.
15. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Lecture Notes in Computer Science*, 2031, 2001.
16. L. Howard Holley and Barry K. Rosen. Qualified data flow problems. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1980.
17. Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, 2001.
18. Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *Proceedings of the International Symposium on Static Analysis (SAS)*, 2003.
19. John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1994.
20. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.
21. Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Proceedings of the 5th International Symposium on Static Analysis (SAS)*, 1998.
22. Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, 2005.
23. K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, 2005.
24. Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *Proceedings of the 9th International Symposium on Static Analysis (SAS)*, 2002.

25. Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, 2003.
26. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, May 2003.
27. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience*, 30(7):775–802, 2000.
28. Madanlal S. Musuvathi, David Park, Andy Chou, Dawson R Engler, and David L Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.
29. Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the ACM Symposium on Principles of programming Languages (POPL)*, 2005.