

Symbolic Path Simulation in Path-Sensitive Dataflow Analysis

Hari Hampapuram, Yue Yang, and Manuvir Das
Center for Software Excellence
Microsoft Corporation

{hhampa | jasony | manuvir}@microsoft.com

ABSTRACT

Symbolic path simulation is becoming an increasingly important component in many static analysis tasks. The emergence of inter-procedural path-sensitive dataflow algorithms has both raised the demands and posed new challenges for effective techniques in path feasibility analysis.

This paper develops a general-purpose path simulator and applies it to support path-sensitive dataflow analysis. The core component of the path simulator is a simulation engine that supports a wide variety of programming language features. This simulation engine can be “wrapped” with an interface layer to support a given client application.

As a concrete case study, we discuss the experiences gained in integrating the path simulator with ESP, a software validation tool for C/C++ programs. We apply ESP to validate a future version of Windows against critical security properties. Our results show that the global path simulation mechanism is both critical in improving precision and scalable enough to be of practical use.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms

Reliability, Verification

Keywords

Symbolic simulation, path feasibility, dataflow analysis

1. INTRODUCTION

A fundamental question arisen in program analysis is the following: Given a path in the Control Flow Graph (CFG) of a program, is it feasible (executable)? Due to the conservative nature of static analysis, a large portion of paths

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASTE '05, September 5-6, 2005, Lisbon, Portugal.

Copyright 2005 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

```
extern int a, b;
void Process(int handle) {
    int x, y;
L1:    if (a > 0) {
        CloseHandle(handle);
        x = 1;
    }
    else
        x = 2;
L2:    if (b > 0)
        y = 1;
    else
        y = 2;
L3:    if (x != 1)
L4:        UseHandle(handle);
}
```

Figure 1: A simplified code snippet adapted from an OS kernel.

considered in static tools are not feasible. Avoiding the exploration of such paths is beneficial in reducing the number of false positives. Thus, an effective technique for improving precision is to harness the analysis with a symbolic path simulator that ensures path feasibility.

Consider, for example, the simplified code snippet adapted from an operating system kernel shown in Figure 1. We intend to check the property that the program should not call `UseHandle` on an object handle that has been previously closed by `CloseHandle`. This code snippet reflects the fact that programmers often use status flags (such as x and y in this example) to record program states. These flags are then checked to guide control flow. Without taking path feasibility into account, an analysis may report a potential violation at L4. A more careful analysis, however, reveals that no valid error path can lead to this point. Hence, the warning is a false positive.

Imprecision poses a major obstacle to the successful adoption of a static analysis tool because figuring out spurious results manually is very costly in engineering resources. Therefore, path feasibility analysis is essential to make a tool practical.

Symbolic path simulation has potential applications in broad areas including defect detection, crash dump analysis, code review, and test generation. Unfortunately, exist-

ing path simulation methods do not offer the flexibility and scalability to support different static analyses.

Path simulation has been pursued in the symbolic execution community. But most symbolic execution systems rely on expensive theorem provers and do not accept programs with pointers. PREFIX [1], a successful defect detection tool, performs per-path simulation and does not support the join operation used by merge-based dataflow algorithms. Model checking tools such as SLAM [2] and BLAST [3] rely on path feasibility analysis during abstraction refinement to rule out spurious counter-examples — each of the counter-examples discovered by model checking is also a single path.

In this paper, we present a path simulation infrastructure that is aimed at handling path-sensitive dataflow analysis for industrial programs. To be practical, it must satisfy several challenging (and sometimes conflicting) goals at once: 1) It should support common language features, 2) it should support advanced analysis techniques such as selective merging, 3) it should be precise enough for most analyses, and 4) it should scale to large programs.

We make the following contributions:

- We present the design of a general-purpose path simulation engine capable of handling complex language features. For example, it can track information at bit level; it allows any number of pointer dereferences to be made; and it allows arbitrarily complex operator expressions to be created.
- We integrate the path simulator with ESP. This enables ESP to employ an effective merge heuristic to retain path-sensitivity without the exponential cost. We use this case study to illustrate how to adapt the path simulator for a given client application.
- We apply ESP to validate critical properties for a version of Windows and demonstrate the practicality of our approach.

The rest of the paper is organized as follows: In Section 2, we overview the techniques of ESP. In Section 3, we describe the system architecture of the path simulator. In Section 4, we present the core simulation engine. In Section 5, we discuss how to adapt the simulator for a given application. In Section 6, we summarize experimental results. We review related work in Section 7 and conclude in Section 8.

2. BACKGROUND OF ESP

Traditional flow analyses are mostly path-insensitive; they lose precision by assuming that all paths are executable. Full path-sensitive analysis such as model checking does not scale well due to the well-known state explosion problem. In recent years, path-sensitive inter-procedural flow analysis such as ESP [4, 5] has shown promising potential in striking a good balance between precision and scalability.

ESP is a validation tool for *typestate* [6] properties. Typestates extend the ordinary types: For an object created during program execution, its ordinary type is invariant through the lifetime of the object but its typestate may be updated by certain operations. ESP allows a user to write a custom specification encoded in a finite state machine to describe typestate transitions. Based on the specification, ESP instruments the target program with the state-changing events. It then employs an inter-procedural dataflow analysis algorithm [7] to compute the typestate behavior at ev-

ery program point. To illustrate, recall the example in Figure 1. During the analysis, patterns such as `CloseHandle` and `UseHandle` are matched and the corresponding events are injected to the CFG. A call to `CloseHandle` would change the typestate to `CLOSED`. Any subsequent call to `UseHandle` would lead to the `ERROR` state.

Inter-procedural analysis is handled through the use of partial transfer functions via *function summaries*. Each function maintains a summary, which maps the dataflow facts from the entry node to the exit node. When propagating the dataflow fact from a call node to the associated return node, the analysis consults the function summary of the callee to get the updated dataflow fact. If it is not available, dataflow is dynamically triggered at the entry of the callee to update its summary. One of the key features of our path simulator is its capability of producing function summaries to support demand-driven inter-procedural analysis.

An important technique in ESP is the *property simulation* method for combining dataflow analysis and symbolic evaluation. The algorithm computes two sets of information: (a) the property state (typestate according to the specified protocol) and (b) the path simulation state. At a merge point in the control flow, if two symbolic states have the same property state, ESP merges the path simulation states. Otherwise, ESP explores the two paths independently as in a full path-sensitive analysis. This method allows ESP to gain path-sensitivity while avoiding the exponential blow up. This selective merging heuristic is effective since it 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.

Example. In Figure 1, the branch from L2 does not affect the typestate property. Therefore, ESP merges the path simulation states that arise from this branch, *i.e.*, it drops the correlation between *b* and *y*. In contrast, the branches at L1 and L3 cause updates in typestate. As a result, ESP tracks those paths accurately. With the assistance from the symbolic path simulator, ESP is able to conclude that the potential error point at L4 is infeasible, thereby validating that the code does not violate the protocol. \square

The above example demonstrated the critical role played by the path simulator. The simulation engine must offer sufficient scalability, efficiency, and reasoning power, since they all directly affect the overall system performance. The need for supporting inter-procedural path-sensitive analysis poses a unique challenge as it requires the simulator to handle join as well as procedure calls, in the presence of complex language constructs. The technical merit of this paper lies in addressing these issues and integrating practical techniques to produce useful results.

3. SYSTEM ARCHITECTURE

The architecture of the path simulator is shown in Figure 2. The system consists of a reusable component called *Simulation State Manager* (SSM) and an interface component called *Simulation Interface* (SI). The Simulation State Manager maintains a set of symbolic states referred to as *simulation states*. It also acts as a theorem prover to answer queries about a state. The Simulation Interface symbolically evaluates the program and sets up the simulation states ac-

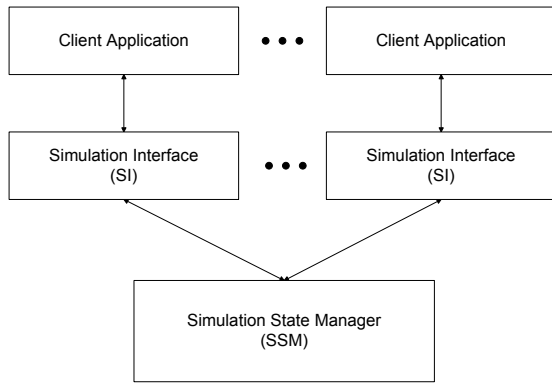


Figure 2: The architecture.

$region \in Loc$
 $value \in Val$

$\rho : Environment = ProgramSymbol \rightarrow Loc$
 $\sigma : Store = Loc \rightarrow Val$

Figure 3: Semantic domains.

cordingly. A client analysis may require its own Simulation Interface but all analyses share a common Simulation State Manager. The Simulation Interface layer isolates the program semantics from SSM, thus making SSM reusable for different applications.

The semantic domains used by the simulator are shown in Figure 3. An *environment* is a mapping from program symbols to locations. A *store* is a mapping from locations to values. The simulator acts as a virtual machine in which the Simulation Interface maintains the environment and the Simulation State Manager tracks the store.

The Simulation Interface can take any suitable program representation as input. Without losing generality, this paper assumes that the Control Flow Graph is available and program symbols are represented as CFG expressions. Thus, the task of the simulator is to track enough semantic information about CFG expressions involved in branch conditions (or *path predicates*) to determine, in a conservative way, if the branch conditions along a path are valid.

4. SIMULATION STATE MANAGER

The purpose of the Simulation State Manager is to provide a memory model of the store that is compatible with C/C++ and extensible to many programming languages. In addition to tracking known values, the model must also be able to track *partial knowledge*. For example, the value of a whole structure may not be known, but one of its fields is known — SSM should be able to reason about this fact.

4.1 Region-Based Abstraction

The Simulation State Manager uses a region-based abstraction to represent the store: *Regions* represent locations; they are mapped to *values* according to the store. Intuitively, a region can be thought of as representing a chunk of contiguous memory allocated for some object. This model is chosen because it is similar to that of C, which is among the least restrictive abstractions.

```

void foo(int *p, int *q) {
    int x = *p;
    int y = *q;
    if (p != q)
L1:     return;
L2:     if (*p != *q) {
        ... // Not reachable!
    }
}

```

Figure 4: An example of different regions.

4.1.1 Regions

A region can be *size-based* or *field-based* to represent scalar type or compound type, respectively. It can be further categorized as a *variable region* or a *deref region*. The former denotes the location of a variable or parameter whereas the latter represents the location pointed to by some pointer. Variable regions have a uniform meaning, *i.e.*, two different variable regions always represent distinct locations and the same variable region from different simulation states represent the same location. In contrast, two deref regions may or may not represent the same memory location.

Example. In Figure 4, the regions created for x , y , p , and q are variable regions; the regions created when evaluating $*p$ and $*q$ are deref regions. Even though these deref regions are created as different regions, the simulator can deduce that they must refer to the same location at L2 because the function would have returned at L1 otherwise. \square

It is useful to distinguish variable regions from deref regions. This is because some operations (*e.g.*, join and binding) need to establish the correlation between two sets of regions. In these cases, comparing variable regions can serve as a starting point and the mapping between deref regions can be populated following pointer reachability.

4.1.2 Values

The Simulation State Manager defines a rich set of values to encapsulate known information at regions. *Constant values* represent numerals (*e.g.*, integers, floats, or doubles). *Operator values* represent expressions involving operators (*e.g.*, arithmetic, bitwise, or relational operators). *Symbolic values* act as constraint variables, which can be used to impose assumptions. A special group of constraint variables, referred to as *region-initial values*, impose constraints on the initial values at a function entry point. A *pointer value* type indicates that a value is a pointer. A value can also be *field-based* to represent a compound type.

It is worth noting that region-initial values are needed to support “lazy association” between regions and values: A region gets its region-initial value by default unless it has been assigned or killed.

A value in SSM differs from the value in real memory in that a value in simulation has an identity, *i.e.*, there is a single instance for any value. This makes it easy to establish relations.

Example. Suppose the code “ $b = a$; if ($b \neq a$) ...” is simulated and the exact value of a is not available. The

simulator can still quickly deduce that “ $b \neq a$ ” must be **false** because both a and b refer to the same value. \square

This paper uses the following notation: A region name starts with **R**; a value name starts with **V**. A CFG expression may appear as subscript for clarity. Symbolic values, region-initial values, and pointer values are denoted as $\$$, **init**, and **ptr**, respectively. For instance, the region for a can be denoted as R_a and its region-initial value can be denoted as $V_{init(R_a)}$.

4.1.3 Input Values and Current Values

Two kinds of information need to be tracked for regions: (a) the initial values when the function is entered and (b) the actual values in the store. To formalize this, we define *input nodes* as global variables and formal parameters and transitive dereference targets of these. Similarly, *output nodes* include all globals, the return variable, and dereference targets of these and the formal parameters.

$$\begin{aligned} \text{Deref}(p) &= q \text{ if } p \rightarrow q \\ \text{reached}(v) &= \{v\} \cup \text{reached}(\text{Deref}(v)) \\ \text{reached}_p(v) &= \text{reached}(v) - \{v\} \end{aligned}$$

$$\begin{aligned} \text{inputNodes}(f) &= \\ & \bigcup_{p \in \text{params}(f)} \text{reached}(p) \cup \bigcup_{g \in \text{globals}} \text{reached}(g) \\ \text{outNodes}(f) &= \text{reached}(\text{return}_f) \\ & \cup \bigcup_{p \in \text{params}(f)} \text{reached}_p(p) \cup \bigcup_{g \in \text{globals}} \text{reached}(g) \end{aligned}$$

Input regions are the regions for input nodes. The values present in the input regions at the time the function is entered are called *input values*. Actual values evaluated according to the store are called *current values*. *Output regions* are the regions for output nodes. Recognizing the difference between these two kinds of values will become important when binding procedure calls.

4.2 Simulation States

A simulation state is a tuple $\langle M, P, K \rangle$ where M is a map that represents the store between regions and values, P is a set of predicates that represent binary relations between values (these predicates are implicitly conjuncted), and K is a kill set that records which expressions have been killed by direct assignment.

The kill set is used for determining whether an expression still has its initial value or has been killed. As previously mentioned, this provides a mechanism to lazily associate region-initial values with input regions. The alternative approach would be to explicitly assign region-initial values to all reachable input locations — this is not efficient, especially considering that there may exist potentially unbounded deref regions.

A simulation state S at program point P in function F captures an approximation to the requirement for all paths that can reach P . Given the predicate constraints on input values and current values, say $C_{in}(S)$ and $C_{out}(S)$, the simulation state S summarizes requirements for a legal path that can reach P : If the path satisfies $C_{in}(S)$ at the entry of F , $C_{out}(S)$ must hold at P . In particular, the simulation state at the exit node of a function captures the summary of the function — this is a key observation for supporting partial function summaries during inter-procedural analysis.

Given a set of feasible paths L , we define a partial order $\sqsubseteq: L \times L \rightarrow \{\text{true}, \text{false}\}$ as $L1 \sqsubseteq L2$ iff $L1 \subseteq L2$. Thus,

the operation of Simulation State Manager can be carried out as a fixpoint computation on the complete lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ where the join operator \sqcup is set union.

4.3 API of the Simulation State Manager

SSM provides two kinds of APIs: one for updating and the other for querying the simulation states.

Updating operations:

- Create, drop, clone, and compare simulation states.
- Create regions and values.
- Set values at regions.
- Evaluate operator expressions for constant values.

Querying operations:

- Get attributes of regions and values.
- Get values stored in regions.
- Find which regions hold pointers.
- Dereference pointers.
- Query the validity of a predicate.

4.4 Handling the Store

Given a region, the Simulation State Manager allows an application to set it with a value via **SetValue** or get the value from it via **GetValue**. **SetValue** updates the entry in the store map M . It also uses the kill set K to record the fact that the definition of the expression associated with the region has been killed. **GetValue** looks up the store map M . If a matched value can be found, it is returned. Otherwise, there are two possibilities: (a) The value of the region has not been assigned, in which case a region-initial value is returned, or (b) the value of the region has been killed, in which case a symbolic expression is returned. The simulation state manager identifies the above cases by checking the kill set.

4.5 Handling Predicates

The Simulation State Manager allows an application to establish facts and make queries about them. In general, there are three kinds of facts — *equalities*, *disequalities*, and *inequalities*. A simulation state maintains sets of equivalence classes. Each equivalence class holds the values that are known to be equal. Disequalities are maintained as multi-maps between equivalence classes. Inequalities are represented as a graph where nodes represent equivalence classes and the edges represent inequality relations. Each inequality relation is assigned with a weight. To answer the query “ $V1 < V2$ ”, the Simulation State Manager looks up the corresponding equivalence classes $E1$ and $E2$ and computes the shortest distance from $E1$ to $E2$ in the graph.

The decision procedure implemented using these data structures allows queries about equalities, disequalities, and inequalities. In addition, it supports a set of simple inference rules for uninterpreted functions and linear arithmetics. The structure of the simulation state is designed such that more sophisticated theorem provers can be called if necessary.

4.6 Handling the Join Operation

The join operation moves the symbolic states upwards in the lattice, *i.e.*, there are less constraints about feasible paths after the merge. Path feasibility analysis is undecidable in general. To guarantee convergence and efficiency, the Simulation State Manager generates a superset from the union of the incoming states, *e.g.*, when dealing with loops. Such over approximation is conservative — it will not miss errors but may introduce imprecision. Our experience suggests that it is a good strategy to start with a light-weight decision procedure and fine-tune it based on the experimental evidence obtained from a given application.

In the join operation, we first map the *memory graphs* between the incoming states and the joined state. A memory graph is a graph where each node is a region and each edge indicates that the value contained in the source region points to the target region. To properly map regions in the presence of pointers, we use a worklist algorithm that starts with all variable regions. As long as the worklist is not empty, we remove a region from the worklist and get its value. If the value is equivalent to a pointer, we add the region pointed to by the pointer to the worklist. This process ensures that all deref regions get populated. After establishing the memory graph, we can join the store maps, predicates, and the kill sets from the incoming states.

4.7 Transactions

Information carried by a simulation state gets updated when certain operations are performed. The sequence of such operations, referred to as *transactions*, completely characterizes a simulation state.

Based on this observation, a simulation state can simply record its transactions instead of carrying the data structures to maintain all facts — those facts can be created by “playing back” the transactions when needed. Only two types of transactions are needed for our purposes: **Assign** and **Assume**. The former is for setting the value of a region and the latter is for establishing an assumption about values.

4.8 Memory Pools

To achieve scalability, the Simulation State Manager uses memory pools. Simulation states are persisted on hard disk when the pools are full; they are read back to memory when needed. When check-pointing simulation states, only transactions need to be written to persistent storage.

5. SIMULATION INTERFACE

For ESP, the purpose of the Simulation Interface is to define a set of transfer functions for operations such as assignments, branches, as well as procedure calls and call returns.

5.1 Fetching Regions and Values

The Simulation Interface takes the CFG expressions as input and updates the simulation states through the Simulation State Manager. Given a CFG expression, **FetchRegion** and **FetchValue** are used to get the region or value from the simulation state. These two functions are defined recursively by each other when pointer dereference is involved.

5.1.1 Fetching a Variable Region

A symbol table is maintained to map the environment between CFG expressions and variable regions. For a variable

expression, it is straightforward to fetch its region by looking up the symbol table. If needed, the Simulation State Manager is asked to create a region.

5.1.2 Fetching a Deref Region

Special care must be taken when dereferencing a pointer. In this case, the CFG expression has a format of “ $*p$ ” or “ $p \rightarrow f$ ”. To fetch the region for $*p$, we first need to fetch the region R_p for p and use **FetchValue** to get its value V_{R_p} . Then we do a case analysis by looking up the equivalence class of V_{R_p} :

- If V_{R_p} is equivalent to a pointer value, say $V_{ptr(R_q)}$ that points to a region R_q , we return region R_q .
- If V_{R_p} is a symbolic value or region-initial value, we create and return a new deref region R_{*p} . To record the fact that this new deref region is pointed to by V_{R_p} , we also create a pointer value $V_{ptr(R_{*p})}$ that points to R_{*p} and add an equivalence predicate to indicate that V_{R_p} is equivalent to $V_{ptr(R_{*p})}$.

Similarly, to fetch the region for “ $p \rightarrow f$ ”, we fetch the deref region R_p for p and return its sub-region at field f .

5.1.3 Fetching a Value

For literal expression, a constant value is returned. For a variable expression x , the Simulation Interface first calls **FetchRegion** to get the region R_x . It then queries the Simulation State Manager to get the value V_{R_x} from the simulation state.

5.2 Assignments

When an assignment “ $lhs = rhs$ ” is encountered, the Simulation Interface gets the region of lhs using **FetchRegion** and the value of rhs using **FetchValue**. Then it calls the Simulation State Manager to assign the value to the region and add an entry to the kill set to record that lhs has been killed.

Example. For the assignment “ $x = 1$ ”, the Simulation Interface fetches the region R_x for x and calls **SetValue** via the SSM to set the value of R_x to be V_1 . \square

5.3 Branches

Given a branch expression “ $lhs \text{ op } rhs$ ”, the Simulation Interface translates the operands and relational operator from CFG expressions to SSM values through **FetchValue**. Then the Simulation State Manager is queried to determine the validity of the branch condition. Depending on the outcome, the Simulation Manager can guide the analysis in two ways: (a) If the answer is **true** or **false**, only the feasible branch direction is followed. (b) If the answer is **don’t know**, both branch directions will be processed. In the latter case, the simulator adds a corresponding predicate to the successive simulation state in each path, recording the assumption made about the branch condition.

Example. Consider the statement “**if** ($a == 0$) { $x = 1$;} **else** { $x = 2$;}” and suppose the value of a is a symbolic value $V_{\$1}$. Because the exact outcome is not known, the simulator explores the true branch with an additional predicate $V_{\$1} = 0$ and the false branch with an additional predicate $V_{\$1} \neq 0$. \square

5.4 Into-Binding

Sometimes it is helpful to pass precise context information into the callee. *Into-binding* is the process that maps the call site simulation state S to the callee entry simulation state S' . The following tasks are performed by into-binding:

- For each parameter, its region-initial value in S' is mapped with the value of the actual parameter in S .
- For each global variable, its region-initial value in S' is mapped with its corresponding current value in S .
- The predicates in S' and S are mapped.

The above operations all involve fetching values from the call site and binding them to the region-initial values in the callee simulation state. As discussed earlier, when a pointer is dereferenced, the simulator also needs to map the value of the region pointed to by the pointer.

5.4.1 An Alternative Approach

Instead of binding precise context information into the callee, an alternative approach is *not* to set any constraints to the entry callee state, *i.e.*, all paths are allowed. When the function summary is applied, an infeasible path can still be refuted if the assumptions carried in the summary state contradict the facts in the calling context. This approach has the advantage that a function summary can be reused in different calling contexts. The drawback is that it imposes a higher demand on the reasoning power of the theorem prover because facts must be deduced in terms of symbolic values as opposed to, in many cases, constant values. Our Simulation Interface supports both methods.

5.5 Back-Binding

Back-binding is the process of mapping the function summary to the return node. The following tasks are performed by back-binding:

- Back-bind the region-initial values of input regions. As additional constraints may be imposed to those region-initial values, it is important to check whether they are consistent with the corresponding values in the calling context. This is especially crucial for refuting infeasible paths when into-binding is not precise.
- Back-bind values of output regions.
- Back-bind predicates. If the predicate only involves the region-initial values, the condition must hold at the call site. We evaluate the mapped predicate at the call site. If it is `false`, the path is refuted. If it evaluates to `don't know`, we add a new predicate to the return site simulation state.
- Back-bind the kill set.

5.6 Handling Aliasing

In a conservative mode, when a definition is killed, all the aliased expressions are also killed. The path simulator allows the client application to register an alias analysis engine. In ESP, a scalable context-sensitive flow-insensitive points-to analysis [8] is used to answer aliasing queries.

5.7 Program Slicing

For path feasibility analysis, a CFG expression is not worth being tracked unless it can affect branch conditions. As a result, a large number of “irrelevant” operations can be ignored through program slicing. To support this, we pre-process the program with the flow-insensitive points-to analysis [8] to find all the nodes that may flow to branches.

6. EXPERIMENTAL RESULTS

ESP has been applied to check the usage of certain APIs that are designed to be “overloaded”. These function interfaces provide various functionalities, sometimes under different privilege modes, based on the bit-field value of the input parameter. If not used properly, however, an implementation using these APIs may be accidentally exposed to malicious attacks.

Checking this kind of property has been difficult with previous tools because it requires precise tracking of value flow in large programs. With the assistance of the path simulator, ESP has succeeded in validating the device drivers of a future version of Windows against this security vulnerability. Using two PCs, each with a 3.06GHz Xeon CPU and 2GB of RAM, the analysis completed in about 20 hours. It reported several hundreds of traces from 149 device drivers. Most of these warnings have since been confirmed and fixed.

We also conducted a comparison analysis between using and not using the path simulator during the analysis. As an example, Table 1 outlines the performance statistics for checking the Windows kernel subsystem. The source program has 216,000 LOC and 9755 functions. When path simulation is applied, ESP reports 2 real bugs and no false positives. When path simulation is not used, ESP also reports 12 spurious errors along with the 2 actual bugs.

	Bugs	False Positives	Time (sec)
With Sim	2	0	1098
Without Sim	2	12	1037

Table 1: Comparison results using one PC with a 3.06GHz Xeon CPU and 2GB of RAM.

These experiments demonstrate that (a) it is feasible to integrate a global path simulator to diagnose real program failures, and (b) path simulation can improve precision significantly with a modest performance cost. Our experience indicates that path feasibility analysis can even lead to performance gain in some cases. This is because the cost incurred by the simulator is often offset by the reduced number of paths that would need to be analyzed without path feasibility analysis.

7. RELATED WORK

Symbolic path simulation can be viewed as an instance of symbolic execution. However, existing symbolic execution systems, *e.g.* [9, 10, 11, 12], employ expensive decision procedures that result in high complexity. Furthermore, they do not provide scalable solutions for tracking complex language constructs such as pointers. We focus on designing a general-purpose and scalable path simulation engine applicable for practical use.

Liu et al. [13] apply parametric regular path queries on graph representation of programs to conduct path analysis.

While their work focuses on a simple and powerful language for expressing analyses, our work focuses on identifying feasible inter-procedural paths in scalable defect detection.

Many software validation tools have relied on path feasibility analysis. ESC-Java [14] is a local analysis tool that applies a theorem-prover called Simplify to verify pre-conditions and post-conditions in annotated Java programs. Our approach is less rigorous. But it supports global analysis and it does not require user annotations. The summary simulation state can be regarded as automatically generated annotations.

Theorem provers such as Simplify [14] apply SAT solving technique and often combine different theories. The Simulation State Manager is designed at a higher abstraction for easy integration with different applications. Off-the-shelf theorem provers can be applied as a plug-in.

SLAM [2] and BLAST [3] apply model checking with iterative refinement. They use a theorem prover to determine the feasibility of the counter examples discovered by model checking. These counter examples are concrete paths. In contrast, our simulator handles more complex paths and supports join.

PREFIX [1] is a successful global software validation tool, which truncates the search space to achieve scalability. Although path simulation plays a central role in PREFIX, the simulation is conducted in a per-path manner. As a result, it cannot support merge-based global analysis. Our framework is aimed at providing more generic support for path simulation.

8. CONCLUSIONS

We have presented the design and application of a symbolic path simulator and demonstrated that our framework is both effective in cutting down false positives and scalable enough for solving real world problems.

Although this paper only discussed the integration effort in the context of ESP, porting the Simulation Interface to another application involves similar techniques. Several applications to date have already gained benefit from the flexibility of the simulator. For example, the simulator has been adapted to support PSE [15], a postmortem static analysis tool. It has also been customized in [16] to provide iterative path refinement for ESP.

The framework outlined in this paper offers a foundation upon which future enhancements can be pursued in orthogonal directions. On the one hand, more language features and different applications (*e.g.*, test generation and interactive simulation) can be supported. On the other hand, the precision of the theorem prover can be improved by integrating advanced proof techniques.

Acknowledgments. We are very grateful to Stephen Adams, Zhe Yang, and Vikram Dhaneshwar for their infrastructure support and insightful discussions.

9. REFERENCES

- [1] 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.
- [2] Thomas Ball and Sriram Rajamani. The SLAM project: debugging system software via static analysis. In *the ACM Symposium on Principles of Programming Languages*, 2002.
- [3] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *the ACM Symposium on Principles of Programming Languages*, 2004.
- [4] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.
- [5] 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*, 2004.
- [6] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.
- [7] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural data flow analysis via graph reachability. In *the ACM Symposium on Principles of Programming Languages*, 1995.
- [8] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *8th International Symposium on Static Analysis*, 2001.
- [9] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezze. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001.
- [10] N Gupta, A Mathur, and M Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1998.
- [11] E Gunter and D Peled. Path exploration tool. In *TACAS*, 1999.
- [12] Jian Zhang. Symbolic execution of program paths involving pointer and structure variables. In *the Fourth International Conference on Quality Software*, 2004.
- [13] Yanhong Liu, Tom Rothamel, Fuxiang Yu, Scott Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.
- [14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.
- [15] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: Explaining program failures via postmortem static analysis. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2004.
- [16] Dinakar Dhurjati, Manuvir Das, and Yue Yang. Path-sensitive dataflow analysis with iterative refinement. Technical Report MSR-TR-2005-108, Microsoft Corporation, 2005.