Identifying Execution Points for Dynamic Analyses

William N. Sumner
School of Computing Science, Simon Fraser University
wsumner@sfu.ca

Xiangyu Zhang
Department of Computer Science, Purdue University
xyzhang@cs.purdue.edu

***Abstract*—Dynamic analyses rely on the ability to identify points within or across executions. In spite of this being a core task for dynamic analyses, new solutions are frequently developed without an awareness of existing solutions, their strengths, their weaknesses, or their caveats. This paper surveys the existing approaches for identifying execution points and examines their analytical and empirical properties that researchers and developers should be aware of when using them within an analysis. In addition, based on limitations in precision, correctness, and efficiency for techniques that identify corresponding execution points across multiple executions, we designed and implemented a new technique, Precise Execution Point IDs. This technique avoids correctness and precision issues in prior solutions, enabling analyses that use our approach to also produce more correct results. Empirical comparison with the surveyed techniques shows that our approach has 25% overhead on average, several times less than existing solutions.**

I. Introduction

Dynamic analyses help developers identify interesting program behaviors at runtime. As a result, these analyses can simplify or speed up common tasks like debugging [1] and verification [2]. Because these problems are core development tasks, improving them can lead to lower software development costs [3], and this, in turn, has led to great interest and growth within the field of dynamic analysis as a whole.

One fundamental task in dynamic analyses is identifying a point within an execution of a program. Such *execution points* are sometimes used to provide feedback to developers [4], [5]. For example, when a tool like Memcheck within Valgrind [4] identifies an invalid memory access, it can provide an execution point showing developers where in the execution this invalid access occurred. Developers can use this information to help fix the bug. Execution points also serve as input to additional analyses. For instance, *dual slicing* uses execution points to identify commonly executed instructions across multiple executions [6]. It uses these common behaviors to prune out irrelevant dependences from specialized slices that concisely explain concurrency bugs.

In spite of this pervasiveness, dynamic analyses are inconsistent and imprecise in how they identify and compute execution points. Analyses create their own formulations of *execution point IDs* (EPIDs) without understanding existing approaches, and even among existing techniques, different types of EPIDs have unexplored properties. Their strengths and weaknesses are poorly understood and can lead to unexpected limitations of precision or scalability. In addition, one definition of execution point may be preferable in one context but undesirable in another, yet these trade offs between the different techniques are presently not well understood.

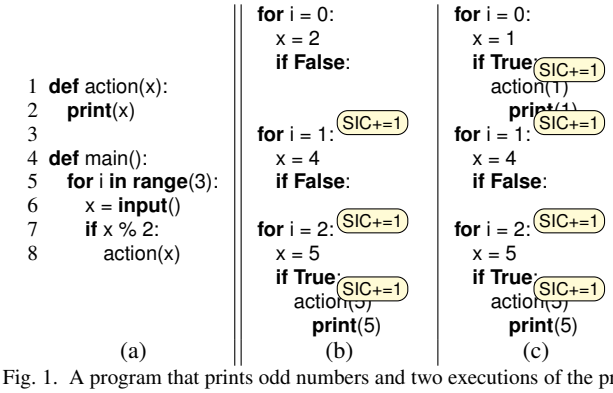

Fig. 1. A program that prints odd numbers and two executions of the program.

Consider the program in Fig. 1a. This program reads in three numbers from the user. If a number is odd, as checked on line 7, then the program calls action() to print the number out. Notice that line 2 can execute many times because it is called from within the loop. As a result, simply using the line number to identify the execution point is *ambiguous* because the same ID may appear multiple times within the same execution. This is undesirable for many dynamic analyses, as it yields imprecise or incorrect results [1], [7].

One approach, commonly used in the context of record and replay techniques [7]–[10], is the *Software Instruction Counter* (SIC). An SIC uses a single integer counter that increments at function calls and loop backedges during the execution of a program. Combining the current counter with the current line number yields a pair (*counter*, *line*) that can uniquely identify an execution point within one execution. For example, the instances of the **if** statement on line 7 of execution (b) are identified by (0,7), (1,7), and (2,7) because of the counter increments on back edges as shown in Fig. 1b.

However, these identifiers only work *within a single execution*. The SICs used for execution (b) do not work for the instructions of execution (c). This is because the executions behave differently. The SIC is incremented at the call to action() in the first iteration of execution (c), so the identifiers for the **if** statements are (0,7), (2,7), and (3,7). Because the SIC was incremented at the first call in (c) but not in (b), the SICs of the two executions diverge and cannot be compared after this point. This is a problem for analyses that compare information *across multiple executions* [1], [6], [11], [12] because SICs can only precisely identify points within one execution.

To address this problem, and enable comparison across

executions, other EPID techniques exploit program structure [13], [14]. Using this information, they are often able to align the corresponding instructions across multiple executions. Unfortunately, these techniques also have limitations that create ambiguous or meaningless relationships when identifying the instructions that align across executions. They also have substantial limitations in usability. In particular, Structural Execution Indexing [13] has difficulties scaling to longer executions, while STAT [14] requires a program core dump at each point that requires an EPID. In addition, SEI can fail to identify useful relationships between EPIDs.

In this paper we survey five existing approaches used to compute EPIDs for dynamic analyses. The surveyed techniques range in their precision and purpose from only being able to imprecisely identify points even in one execution to uniquely identifying points across multiple executions even in the presence of concurrency and nondeterminism. They range in runtime overhead from none, using only postmortem analysis, to several times the cost of the original execution. Based on the limitations of the existing techniques for cross-execution EPIDs, we observed a need for a new technique that provides meaningful and unambiguous relationships in execution alignment and without the usability and scalability limitations of existing approaches. We introduce a new technique for *Precise Execution Point IDs* (PEPIDs) that addresses these goals and has a runtime overhead only slightly higher than using calling contexts [15].

We have implemented all of these techniques, those surveyed along with PEPID. We evaluated them empirically on SPEC CINT2006 to illustrate their performance. We also provide the first analytical comparison of these different approaches, weighing their costs, their benefits, and the scenarios where one may be more desirable than another. Using this information, a dynamic analysis designer can know in advance which techniques are most appropriate for his or her purposes and avoid inventing or reinventing an approach with known problems. In summary, the contributions of this paper are:

1) We surveyed and implemented existing techniques for computing EPIDs for dynamic analyses. We analytically examine all of the different techniques and compare them along several spectra in order to weigh their relative costs, merits, and limitations.
2) We observed problems with producing meaningful, unambiguous relationships between EPIDs as well as with usability and scalability in existing techniques for cross-execution EPIDs. To address these problems, we introduce a new cross-execution technique (PEPID) and show that it avoids the limitations of existing work while also having lower runtime overhead.
3) We empirically compare the runtime and space overheads introduced by the different techniques, those surveyed as well as PEPID, and we show that for cross execution EPIDs, PEPID is the most efficient with 25% average overhead. For intra-execution techniques, SICs are the most efficient, with 9% overhead.
4) We illustrate how missing meaningful relationships between inter-execution EPIDs can result in undesirable or incorrect results for dynamic analyses.

II. Existing EPID Techniques

In this section, we review the different approaches for computing EPIDs that have appeared historically in the context of dynamic analyses. We consider the intended use cases, design, and requirements of each technique.

A. Calling Contexts

One of the traditional representations of EPIDs is the *calling context* at a point within an execution. The calling context consists of the list of active functions currently on the call stack. Note, similar to using the line number or program counter as an EPID in Fig. 1, the calling context is an ambiguous representation. The same calling context may appear multiple times even within one execution. As a result, calling contexts are potentially ambiguous EPIDs, but they provide a more detailed representation of the static program behavior than just a line number. In spite of this, calling contexts are already familiar to developers and can be easily collected by walking over the call stack [4], [16]. As a result, many dynamic analysis tools use calling contexts during analysis or while generating reports for developers [4], [5], [17], [18].

In spite of their familiarity, calling contexts were traditionally costly to collect. Walking over the call stack at every point of interest can be costly, which has forced some dynamic analyses to resort to sampling techniques that only analyze portions of an execution [19]. More recently, efforts have focused on efficient means of encoding calling contexts. These include approaches that can probabilistically encode contexts in constant space [20]–[22] as well as approaches that can precisely encode calling contexts but can require a flexible encoding size and a slightly higher runtime overhead [15].

In the context of dynamic analysis, more precise information is usually preferred, so in this paper we only consider the latter work, *Precise Calling Context Encoding* (PCCE) [15]. PCCE works on the principle that calling contexts are equivalent to paths through the call graph of a program. The technique examines the call graph during compilation and numbers all of the acyclic paths present in the graph. It then annotates every edge, or call site, with an arithmetic operation that computes the numerical ID of the current path in the call graph at runtime, similar to Ball-Larus path profiling [23]. Combined with the current instruction, these comprise a calling context. PCCE handles recursion by pushing and popping the acyclic path IDs onto a calling context stack as necessary.

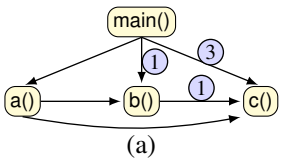

Fig. 2. (a) An annotated call graph that encodes all calling contexts into unique integers. (b) Example instrumentation of the function b().

For example, consider the call graph presented in Fig. 2a. The circle annotations on the call edges denote the amount

added to the context ID before each call and subtracted from the ID upon return. Fig. 2a illustrates this instrumentation for the function b(). Using this example, the calling context main→b→c is captured by the pair (c,2). This reflects the currently executing function, c(), and the numerical ID representing the path in the call graph, 2.

Computing these IDs using PCCE requires that a program be instrumented at compile time, which requires forethought and time not applicable for all dynamic analyses. For instance, if a developer wishes to analyze an already compiled program with a tool that uses PCCE, they have to compile the program again to have the necessary instrumentation added. In addition, the efficiency results achieved by PCCE, 1-3.5% runtime overhead, exploit profile guided instrumentation and additional optimizations for compressing repetitive and recursive calling contexts. Both of these requirements can be avoided by using stack walking to extract the calling context, but, as mentioned before, they induce a high overhead [20].

B. Software Instruction Counters

Mellor-Crummey and LeBlanc introduced Software Instruction Counters (SICs) to provide a more precise notion of execution point for profiling and debugging [24]. SICs have since been used in a variety of dynamic analyses, especially in the context of nondeterministic recording and replay [7]–[10], [25]. SICs provided the first representation of execution points that was able to uniquely and scalably identify every instruction within a single execution of a program. They work by maintaining a monotonic counter that indicates the progress through an execution. This gives SICs the advantage of only adding a single counter and sparse increment operations to an execution, thus yielding low overhead. While the EPIDs defined by SICs are unambiguous within a single execution, the SIC for a point may change across different executions, and the same SIC may even represent different execution points in two different executions as seen in Fig. 1 and Section I.

Computing SICs involves incrementing a counter at every function call and back edge in the control flow graph (CFG) of a program. Fig. 1b-c show the executions of Fig. 1a with instrumentation for computing SICs (where **print**, **range**, and **input** are built-in commands). For any point within an execution of a program, the SIC instrumentation creates a pair, (*counter*,*current line*) such that the pair uniquely identifies that execution point. The counter maintains a notion of forward progress within the execution, and it is only incremented at those features within an execution that may cause an instruction to execute multiple times (loops and function calls).

Accurately placing instrumentation on back edges requires static analysis or some additional dynamic analysis to detect loops within individual functions. This mandates either forethought for the static analysis, just like PCCE, or additional runtime, space, and complexity overhead for dynamic loop detection. Instead of instrumenting back edges in the CFG, many analyses alternatively instrument the branch points within a program [9], [10]. For executions that terminate or have side effects, these are equivalent and have the advantage that branch instructions can be easily identified and instrumented by dynamic instrumentation or virtualization tools [4], [26].

```
1  def action(x):
2    print(x)
3
4  def main():
5    for i in range(3):
6      push((5, 14))
7      x = input()
8      if x % 2:
9        push((8, 13))
10       push((11, 12))
11       action(x)
12       pop(12)
13     pop(13)
14   pop(14)
```

(a)

Possible Calls to action():

$\langle(5,14)(8,13)(11,12),11\rangle$

$\langle(5,14)(5,14)(8,13)(11,12),11\rangle$

$\langle(5,14)(5,14)(5,14)(8,13)(11,12),11\rangle$

(b)

Fig. 3. (a) The program from Fig. 1 instrumented for computing SEIs. The consecutive pushes at 9 and 10 are discussed in the text below. (b) EPIDs for potential calls to action().

C. Structural Execution Indexing

While the EPIDs provided by SIC suffice for *intra-execution* analyses, we saw in Section I that an EPID defined by SIC might correspond to the first iteration of a loop in one execution and the last iteration of a loop in another execution. Indeed, the alignment that SICs create between the instructions of two different executions can match instructions at the *beginning* of one execution with instructions at the *end* of a second. For dynamic analyses that perform *inter-execution* analysis, e.g. execution comparison, this can lead to meaningless results. Intuitively, when there is no relationship between instructions with the same EPIDs across the two executions, comparing them is uninformative.

This provided the motivation for *Structural Execution Indexing* (SEI) from Xin et al [13]. They observed that some dynamic analyses compare execution points across executions, but the way that analyses identified execution points led to meaningless correspondences, like those established by SIC in Section I [11], [13]. They instead sought to use the semantic structure of underlying programs to determine which program points corresponded. They observed that the control structures of a program along with the dynamic control dependence [27] at runtime established a semantic identity for execution points even across different executions, so they used these to uniquely identify instructions at runtime. The technique maintains a stack that keeps track of the currently active control structures while a program executes. This stack then acts as the EPID.

The process of computing an SEI based ID for an execution point is similar to manually maintaining a call stack at runtime, except that dynamic control dependence information is also included in the stack. At every branch (or call) instruction, the instruction ID is pushed onto an indexing stack along with the ID of its postdominator (or return instruction). This *(ID, postdominator)* pair identifies the region of code that is control dependent upon the branch (or call) instruction. Upon encountering a postdominator (or return), all entries in the stack postdominated by that instruction are popped from stack. Applying this process to the code from Fig. 1 yields the new

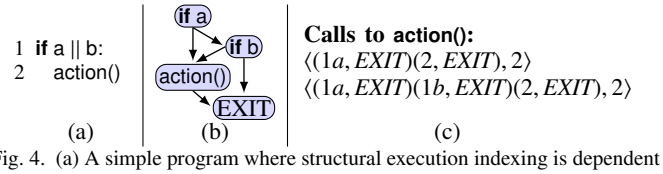

Fig. 4. (a) A simple program where structural execution indexing is dependent upon the execution path. (b) The CFG with two paths to action(). (c) EPIDs for the call to action().

program in Fig. 3a. Note, for each dynamic iteration of the loop, an *(ID, postdominator)* token is pushed on the stack at line 6. As seen in Fig. 3b, showing the EPIDs for each call to action(), these tokens track the monotonic progress of an execution through a loop until the loop finishes and all iteration tokens are popped at line 14. The pop(x) operation removes all tokens with the postdominator x. The push and pop on lines 9 and 13 bound a region of code control dependent upon line 8 [27], while those on lines 10 and 12 identify the call on line 11. Uniquely identifying function calls is crucial because the same function may be called multiple times, and not differentiating the call sites would lead to ambiguous EPIDs.

The complete algorithm also contains additional operations for optimizing simple loops using counters and for eliminating pushes onto the stack that can be inferred based purely on where an instruction lies within the CFG. For example, the push for each loop iteration on line 6 can be replaced by a counter increment, since the loop has a single conditional guard. Also, executing the body of the **if** statement in Fig. 3 automatically implies that the **if** statement on line 8 was executed and the **True** branch taken. Thus, the pushes and pops on lines 9 and 13 can be safely elided.

The intuition that control dependence creates a semantic relationship across executions had previously been used for trace similarity metrics [28] and has proven effective enough that SEI has gained traction in analyses that examine *inter*-execution relationships. It has since been used for tasks ranging from automated debugging [1] to concurrent profiling [29] to identifying causes of vulnerabilities [12]. In spite of this, tracking control dependence can require $O(N)$ space where N is the length of an execution, which does not scale for some programs. The aforementioned optimization heuristics can mitigate this problem in practice, but they do not eliminate it. We explore the space and runtime overheads in Section V. In addition SEI requires that a program be instrumented at compile time to accurately identify postdominators.

As previously noted, SEI was designed to guarantee EPIDs across executions could only be equal for execution points that correspond across the executions. In some cases it is too aggressive in achieving this goal and can create different EPIDs *even when execution points meaningfully correspond across executions*. Consider the code in Fig. 4a. A short-circuiting **or** operation creates the CFG in Fig. 4b with two branches and two paths to action() on line 2. Note that the paths through the program split based on the values of a and b, but the paths that call action() merge together again before this call. Intuitively, the calls to action() occur at the same execution point even in different executions, so their EPIDs should be the same. In spite of this, because SEI bases EPID construction on the control dependence of the execution point, *different paths to the same point can have different EPIDs*. In this case, as Fig. 4c shows, one EPID encodes a path where a is **True** and the call is control dependent on $1a$. The other encodes the path where just b is **True**, and the call is depends first on $1b$ which transitively depends on $1a$ [27]. We show later in Section V-C that this counterintuitive relationship leads to undesirable results for dynamic analyses.

D. STAT Ordering

The techniques presented thus far all required either static or dynamic program instrumentation. In some cases, such as when analyzing a deployed program or a program whose behavior changes when it is instrumented, it is necessary to avoid any instrumentation whatsoever. This motivated the EPID technique presented by Ahn et al. as a part of their *Stack Trace Analysis Tool* (STAT) [14]. STAT was designed for debugging high performance computing applications with multiple processes. In order to better classify and group equivalent processes that represented failures, they developed a technique for analyzing core dumps of programs in order to extract the execution point where a program failed. These core dumps are essentially snapshots of program memory and contain not only the call stack of the execution at the point of failure but also the values of all variables on the stack or heap at that point. In addition to producing an EPID from the core dumps, STAT produced a partial ordering of execution points across different executions. This partial order was particularly important in the context of analyzing parallel code that involved multiprocess communication. STAT was the first EPID technique we are aware of that observed how a partial ordering of EPIDs could be useful for analyses.

EPIDs produced by STAT are also stack based, similar to those produced by SEI. However, STAT does not have the control dependence or postdominance information used by SEI. Instead, STAT infers as much as possible about an execution point from the core dump. In particular, EPIDs from stat interleave (1) the call stack of the execution point and (2) values of certain local variables that show the monotonic progress of an execution through loops. The call stack of an execution point can be extracted from the core dump using stack walking methods mentioned in Section II-A, but finding variables that show loop progress is more difficult. Such variables do not even always exist, so STAT makes no guarantee that EPIDs it produces are unambiguous. Pragmatically, STAT defines *loop order variable*s (LOVs) that can easily be recognized and extracted as indicators of loop progress when present. LOVs must (1) be defined at least once each iteration, (2) be given strictly increasing or decreasing values over a loop's lifetime, and (3) be given an identical value each particular iteration across all possible executions. Informally, these variables are given a strictly ordered and predefined sequence of values. STAT also defines a static analysis for identifying when these variables are available.

Consider the simple program in Fig. 5a. This program contains two loops, one that iterates over a fixed range of

```
1 for i in range(3):
2     process_int(i)
3 for node in linkedList:
4     process_node(node)
```

(a)

Possible Calls to process_int() and process_node():

$\langle(1, \text{i} \mapsto 0) \rightarrow (2, \text{process\_int})\rangle$
$\langle(1, \text{i} \mapsto 1) \rightarrow (2, \text{process\_int})\rangle$
$\langle(1, \text{i} \mapsto 2) \rightarrow (2, \text{process\_int})\rangle$
$\langle(4, \text{process\_node})\rangle$
$\langle(4, \text{process\_node})\rangle$

(b)

Fig. 5. Example of using STAT to identify EPIDs at the calls to process_int() and process_node().

```
1 while a:
2     ...
3     if b: break
4     ...
```

(d)

Fig. 6. A loop with linear SEI growth.

integers on lines 1 & 2 and another that iterates over a linked list on lines 3 & 4. Suppose that the linked list contains two elements. The EPIDs computed by STAT for each call to process_int() or process_node() are shown in Fig. 5b. For the first loop, STAT is able to identify that i is a LOV, so its value inside the loop is extracted and included in the EPID of each function call. This makes the EPID for each call to process_int() unique. However, for the second loop, there is no LOV, as the loop iterates over a linked list. As a result, the EPID contains only the call to process_node(), and the EPIDs are ambiguous.

In contrast to previous techniques, STAT does not require program instrumentation and thus does not induce additional overhead on an analyzed application. However, it can only extract an EPID at a location where the program produced a core dump, e.g. a crashing failure. In practice, this meant that STAT was strictly a post-mortem technique; it could not produce EPIDs on the fly as a program was executing. While this limitation can be worked around by explicitly producing core dumps, both the runtime and space overhead of producing core dumps can be prohibitive. Also note that STAT makes use of static analysis for identifying the LOVs whose values it captures. Performing this analysis precisely requires access to the CFG and variable information available at compile time, but it can also be approximated through binary static analysis, thus avoiding the need for any compile time information.

E. Lightweight Execution Indexing

While SEI offers an approach for computing EPIDs online at runtime, the potential overhead can cause scalability problems and interfere with the program being analyzed. This occurs when loops have multiple guarded exits. Consider the loop in Fig. 6. SEI pushes a token onto the stack every time lines 1 or 3 execute because they branch the control flow, but those tokens will not be popped off the stack until the loop finishes because the branches are postdominated by a statement outside the loop. In order to avoid the overhead of SEI, some analyses instead use information about the number of times an instruction has been seen within a particular calling context, a particular function invocation, or invocations at a certain depth of the call stack [30], [31]. A canonical example of this is *Lightweight Execution Indexing* (LEI), which was used to identify allocated objects in order to help expose potential deadlocks in concurrent Java programs [30].

The approach of LEI is to maintain a counter for each depth of the call stack. This counter keeps track of how many times a particular method has been called at that depth. For instance, the first time that the method foo() is called at a depth of 3 on the stack, its hit counter for the depth 3 is 0. The next time it is called at the depth of 3 on the stack, its hit counter for that depth is 1. The counter for each method at each depth is maintained independently. The LEI for a given execution point then comprises the current calling context along with the hit counts of every call site within the context as well as the hit count and identity of the currently executing statement.

This approach bounds the size of the an EPID to twice the size of the calling context. In addition, it maintains a notion of forward progress through depth counters, and this notion of progress is structured by the call stack. As a result, each EPID is unique and unambiguous within one particular execution. Unfortunately, exactly as with SIC, the values of counters seen in one execution have no guaranteed relationship with the counters seen in other executions. As a result, Lightweight execution indexing can provide EPIDs within one execution, but it cannot provide meaningful EPIDs across executions.

Also similar to SIC, LEI does not inherently require that a program be analyzed or rewritten at compile time. The counters associated with each function and statement of interest at every depth of the call stack can be entirely constructed using dynamic instrumentation without a need for prior planning.

III. Precise Execution Point IDs

Dynamic analyses comparing multiple executions are increasingly common [1], [6], [12], so having a robust, efficient EPID technique that works across executions is important. Such *inter-execution* techniques create EPIDs that are only equal when their corresponding execution points are equivalent. Prior work has called this the *execution correspondence* criterion [13]. In spite of this problem's importance, we see that there are only two existing techniques that can provide EPIDs across executions: SEI and STAT. Both techniques have limitations that can prevent them from being practical or useful for particular dynamic analyses. In particular, we desire an inter-execution EPID technique that is:

- **Online** - An analysis should be able to construct the EPID for the current point in the execution and as often over the lifetime of an execution as necessary.
- **Low Overhead** - An execution running with an EPID technique should require as little additional runtime and memory as possible.
- **Scalable** - Neither the duration of an execution nor the size of its workload should significantly affect the runtime or space requirements of the EPID technique.
- **Unambiguous** - Every instruction or statement within an execution should have a unique EPID.
- **Comprehensive** - As a dual to satisfying the execution correspondence criterion, equivalent execution points should also yield equal EPIDs.

Neither SEI nor STAT is able to satisfy all of these requirements. SEI is not low overhead, scalable, or comprehensive,

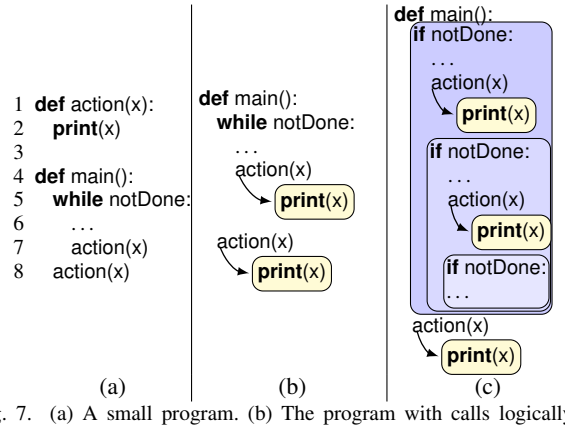

Fig. 7. (a) A small program. (b) The program with calls logically inlined. (c) The program with calls inlined and loops unrolled.

and STAT is not online, unambiguous, or comprehensive. In this section, we introduce a new EPID technique, *Precise Execution Point ID*s (PEPID), that targets all of these criteria. We start by building an intuition about which points *should* correspond across executions in order to provide unambiguity and comprehensiveness. We then devise a technique for computing EPIDs that produces this correspondence efficiently online.

A. Which Points Correspond?

Because we desire an *inter*-execution EPID technique, we must first decide which execution points should correspond or align across executions. The intuition used by SEI was that the path taken by an execution helped to determine which execution points were equivalent, and SEI used control dependence to codify this relationship. STAT, in contrast, used the intuition that loop control variables captured a notion of forward progress through the loop iterations of an execution. But, as we saw before, control dependence prevents comprehensiveness, and focusing on loop control variables leads to ambiguity. In contrast, we base PEPID on the idea that *execution points at the same position in a sufficiently inlined and unrolled CFG are equivalent*.

Consider a simple program with an acyclic CFG and no function calls. Each instruction inside the program can be executed at most once, so an instruction's position within the CFG can unambiguously identify the instruction within an execution. In addition, the same instruction will trivially have the same EPID across all possible executions, thus guaranteeing comprehensiveness. Unfortunately, this model is unrealistic in general; real programs have both function calls and back edges in their CFGs, both of which can cause instructions to execute more than once and thus introduce ambiguity. However, we can extend the intuition of equivalent points in the CFG to handle those cases as well.

First, consider programs that also include function calls. A function may be called from multiple locations, thus executing its body multiple times and making the CFG location an ambiguous EPID. A simple solution to this in most cases would be to inline every function call. If every call were inlined, then function bodies would be duplicated at every call site, once again ensuring uniqueness. Thus, the position of an instruction within this *fully inlined* CFG serves as an unambiguous EPID (ignoring loops). This can be seen in Fig. 7a-b. This simple program makes calls to action() both inside and outside of the loop. Using the position in the CFG alone would make these calls to **print**() on line 2 ambiguous, however, once action() is inlined, the calls from inside the loop are clearly distinguished from those outside of the loop. Of course, this cannot be done in practice because (1) recursive calls would require an undecidable degree of inlining and (2) inlining every function call would simply increase a program's size too much to be pragmatic. However, we only need to perform this operation *logically* for now. We shall later show that the same correspondence can be computed without actually inlining any functions at all.

Next, we must handle back edges in the CFGs of a program's functions. Back edges create loops or general cycles in a CFG and can thus cause instructions to execute multiple times, again making an instruction's position in the CFG ambiguous as an EPID. One approach used by bounded model checkers is to *unroll* the loops of a program [32], [33]. Each iteration of a loop is peeled of into the guarded body of an **if** statement, and each successive iteration is nested within the body of the preceding iteration. Fig. 7c illustrates this unrolling in combination with the inlining of function calls. Again, unrolling a loop sufficiently for all executions is not possible in practice, but we shall show that this limitation is irrelevant in the next section.

Using this combination of unrolling and inlining, we are able to define how execution points relate across executions:

Definition 1 (Alignment): Given two execution points, p_1 and p_2 from executions e_1 and e_2 of program p respectively, let G be CFG of p sufficiently unrolled and inlined to contain both execution points. Points e_1 and e_2 align iff they occur at the same instruction in G.

This *alignment* of execution points determines exactly which points are equivalent and must have equal EPIDs even across different executions. Observe, in this transformed program G, execution points p_1 and p_2 can each be performed at most once in any execution, as guaranteed by the acyclic structure of the unrolled and inlined CFG. Thus, the transformed program guarantees that the position in the control flow graph of the program provides an unambiguous EPID, and the control flow graph correspondence maintains comprehensiveness as before. This means that PEPID avoids the problems with SEI presented in Fig. 4 and Fig. 6.

B. Efficiently Computing PEPIDs

As discussed in the last section, inlining all function calls and unrolling all loops is impractical and even undecidable in general, so we must compute this equivalence another way. Instead of actually performing these program transformations, PEPID executes the original program without any extra inlining or unrolling but at the same time keeps track of the inlining and unrolling operations *that would have occurred* in order to identify the current execution point. We keep track of these

INSTRUMENT(P)
Input: A program P
  for each loop l in P **do**
    insert pushLoopCounter before the loop header of l
    insert incrementLoopCounter before loop latches of l
    insert popLoopCounter on loop exits of l.
  for each call c in P **do**
    insert pushCallSiteID before c
    insert popCallSiteID after c

Fig. 8. INSTRUMENT takes in a program P and modifies it to maintain a PEPID online. This is the unoptimized instrumentation.

operations on an ID stack, similar to those used in SEI and STAT. This stack is then what PEPID uses to produce EPIDs.

In particular, we push an entry onto the stack to identify the call site of every function invocation, popping it as the function returns (or unwinds for exceptional control flow). This tracks the inlining operations for all function invocations. We also need to track all unrolling operations for backedges. We first consider only natural loops, loops with a single entry node or *loop header*, but we extend this to irreducible loops in the next section. We compactly record the unrolling of natural loops by pushing a counter for the loop upon loop entry and popping the counter upon loop exit. We increment the counter upon every iteration of the loop by instrumenting the loop latches, or the edges in the CFG that lead back to the loop header. The stack also naturally handles nested loops.

Fig. 8 shows a naïve instrumentation algorithm for PEPID. It does not cover exceptional control flow, but we handle exceptions by saving the ID stack height before a call that might throw an exception and pruning the stack to that height if an exception was thrown. Note that the entries in the stack related to inlining and the entries related to unrolling may be maintained independently because they can be unambiguously recombined. This stems from the fact that, given an instruction i, the number of static loops containing i may be readily identified. As a result, a PEPID can be broken down into (1) the inlining ID stack, (2) the unrolling ID stack, and (3) the current instruction ID. Observe, though, the inlining ID stack is precisely equal to the calling context! PCCE already provides a means of encoding the calling context that is more efficient than explicitly pushing and popping at each call site, so we can exploit this to make PEPID computation more efficient. At any point during the execution, a dynamic analysis can call GETCURRENTPEPID() to yield an EPID of the form:

$$\langle \textit{PCCE context, unrolling ID stack, current instruction} \rangle$$

This tuple comprises an EPID that provides comprehensiveness and uniqueness based on the prior construction.

Like SEI and STAT, PEPID requires compile-time knowledge about a program. Efficiently computing PCCE calling contexts requires the call graph, and the unrolling stack requires loops to be identified. For programs with only natural loops, PEPIDs are compact. The PCCE context is bounded in size by the calling context depth, and the loop unrolling stack is bounded by the number of nested loops that may be active at one time. We show in Section V that this instrumentation scheme allows PEPID to scale with low overhead.

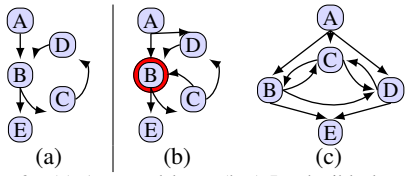

Fig. 9. (a) A natural loop. (b-c) Irreducible loops.

C. Handling Irreducible Loops

Counting iterations is effective for natural loops, which have a unique headers or entry nodes. In that context, unrolling loops is well defined and corresponds to actions upon the unrolling ID stack. Programs can also have unnatural or *irreducible* loops, which have multiple entry points. Indeed, half the SPEC CINT2006 benchmarks have such loops. Fig. 9 shows some natural and unnatural loops. With multiple headers, distinguishing a loop body from a nested loop is difficult. We use Steensgaard's generalized loop forest recognition to identify irreducible loops and their bodies [34][1]. Both (b) and (c) are individual (irreducible) loops under this approach with headers B and D for (b) and B, C, and D for (c).

Sometimes, using an iteration counter can still work for irreducible loops. Given a loop, if there exists a header h of the loop such that every path from each header h' through the loop body back to h' must pass through h, then we say that the header h *naturalizes* the loop. This is because there exists a traversal of the CFG such that every backedge in the loop has h as its destination. Thus, we can use a counter as before and simply increment it on every loop edge that targets h. An alternative intuition is that breaking only edges to h would destroy all cycles in the loop, so a counter incremented on h will uniquely identify instances of this acyclic subregion. Node B in loop (b) is one such naturalizing header. Note that this is just a generalization of natural loops, where the unique header always naturalizes the loop body. We identify naturalizing headers using simple static analysis.

Without a naturalizing header, edges to *multiple* headers must increment the counter to avoid ambiguity. Conservatively, *all* headers may need to increment. This can yield unintuitive results. For example, the path ABDCDCDC in loop (c) would have the EPID $\langle \text{Entry}, \{6\}, C \rangle$ if edges to header nodes increment the loop counter, but so would the path ABCBCBDC. Here, Entry is the calling context, and $\{6\}$ is the unrolling ID stack. Technically, there exists an unrolling of (c) that produces these IDs, but it is unclear how meaningful this is in practice. Alternatively, we can use the same approach as SEI for only this small portion of the program. We push the IDs of predicates in the loop that the headers are control dependent upon and pop them upon their postdominators. This produces the EPIDs $\langle \text{Entry}, \{(B,E)(D,E)(C,E)(D,E)(C,E)(D,E)\}, C \rangle$ and $\langle \text{Entry}, \{(B,E)(C,E)(C,E)(C,E)(B,E)(D,E)\}, C \rangle$, which show the different paths. Both approaches produce unambiguous inter-execution EPIDs. They merely use different approaches for unrolling degenerate irreducible loops. In fact, an analysis can correctly select either. If overhead is more important,

[1]Steensgaard's approach is preferable to other loop extraction techniques in that it produces consistent results regardless of how a CFG is traversed [35].

incrementing on edges to all headers is preferable. If disambiguating paths through these loops is important, then using the localized pushing and popping from SEI is preferable.

IV. Analytical Comparison

In this section, we examine some of the analytical properties of the different techniques surveyed and how they impact which techniques are preferable in different situations. Fig. I summarizes the results, and we discuss them in detail below.

Availability- Many dynamic analyses require that EPIDs be available online, e.g. for identifying events like allocation or synchronization during an execution. Most of the techniques provide EPIDs online, although STAT does not. However, for analyses that are interested in execution points at the point a program crashes, STAT can still be a useful choice because it alone avoids the need for any program instrumentation.

Requirements & Instrumentation- The requirements and time of instrumentation for the techniques can sometimes create more work for analyses or developers that depend on EPIDs. For example, STAT places the lowest instrumentation burden on users and client analyses because it does not modify the underlying program. As a result, it is easy for STAT to be used with an already compiled program. Because it imposes no overhead, it could even be used on deployed software. Techniques like SIC and LEI that use local counters can be implemented using runtime instrumentation alone, so they also impose little burden on users, but they may not be appropriate for deployed software. Finally, the remaining techniques all require that programs are recompiled with additional static instrumentation. This requires the most work and planning on the part of the developer or client analysis.

Independent of instrumentation, the techniques can also require additional source level information to be precise. PCCE, SEI, and PEPID all require additional compilation information, which is expected since they also require static instrumentation. However, STAT also requires some compile time information in order to identify LOVs. This requirement holds in spite of the fact that STAT performs no instrumentation.

Ambiguousness & inter-execution IDs- Ambiguous EPIDs do not necessarily confer much information about where an execution point occurs temporally. Thus, ambiguous techniques may be useful for attaching a lightweight notion of local execution context to an execution point, but they cannot be used for more fine grained execution comparison based techniques [6]. Note, though, that while both PCCE and STAT are listed as ambiguous, STAT is unambiguous for programs in which all loops have identifiable LOVs (hence the '*' in the table).

The major differentiating feature of inter-execution techniques is that they are able to align loop iterations across different executions. As a result, techniques that do not track the progress through each loop independently are unable to provide inter-execution IDs. This effectively leaves only SEI and PEPID as viable techniques for analyses requiring such EPIDs. Note, however, that STAT can also provide this under the same assumptions of LOVs as before (*).

Comprehensiveness- One of the large limitations of SEI was that it was not comprehensive. While its EPIDs always established a correspondence across executions, it also created different EPIDs for execution points that did correspond (see Fig. 4). Note, for programs with LOVs (*), STAT actually *is* comprehensive. However, in contrast to both, PEPID provides comprehensive inter-execution EPIDs in general, making it a preferable choice when instrumentation is possible.

Ordering- Some analyses require that EPIDs be ordered. For example, record and replay techniques require that EPIDs be ordered within one execution (intra) [8]. Some analyses require stricter orders, where EPIDs are partially ordered even across executions (inter) [1], [6], [14]. Most of the techniques are able to provide intra-execution ordering among EPIDs, except for calling contexts with PCCE. SEI, STAT, and PEPID provide stronger inter-execution ordering as well through happens-before relationships among their EPIDs [36].

Space overhead- The size of EPIDs is also an important concern. The required space ranges from none or a constant word, STAT and SIC respectively, to proportional to the length of an execution in the worst case for SEI. All other techniques, however, have EPIDs that grow roughly proportional to the size of the call stack. We examine later how the sizes of the EPIDs produced by these techniques compare in practice.

V. Empirical Evaluation

In order to compare these different EPID techniques in practice, we implemented all of them using LLVM 3.2 as a program instrumentation platform and compared them on the SPEC CINT2006 benchmarks. The implementations cover all basic program behavior covered by these benchmarks, including exceptional control flow. In this section, we look closely at the compile time properties as well as the runtime and space overheads induced by these techniques. We conclude by looking at a particular case study that illustrates why comprehensiveness is important in practice.

Note that neither the runtime nor space overhead comparisons include STAT. This is because STAT performs no instrumentation and thus has no overhead. However, the effectiveness of STAT depends heavily on the ability to produce core dumps and identify LOVs. To gauge whether or not these variables can be found in practice, we compiled the SPEC benchmarks and counted the total number of static loops as well as the number of static loops for which a LOV could be identified. Fig. II contains the results.

Overall, a median of 34% of loops had identifiable LOVs across the different benchmarks, and 31% of all loops had such variables. This indicates that relying on LOVs may not be practical in general. However, STAT was originally designed for analyzing high performance computing programs. For programs in that domain, the structure of the programs may make relying on LOVs practical [14].

A. Runtime Efficiency

For each of the techniques except STAT, we ran the SPEC CINT2006 benchmarks using 'reference' workloads 5 times

TABLE I
ANALYTICAL PROPERTIES OF THE DIFFERENT EPID TECHNIQUES.

Properties	PCCE	SIC	SEI	STAT	LEI	PEPID
Availability	online	online	online	offline	online	online
Requirements	Call Graph	None	Control Dependence Loops	Loop Order Variables	None	Call Graph Loops
Instrumentation	static	dynamic	static	none	dynamic	static
Ambiguous	yes	no	no	yes*	no	no
Inter-execution	no	no	yes	no*	no	yes
Comprehensive	no	no	no	no*	no	yes
Ordering	none	intra	inter	inter	intra	inter
Space Overhead	O(call stack)	O(1)	O(path length)	none	O(call stack)	O(call stack + unrolling stack)

TABLE II
LOV IDENTIFICATION FOR SPEC CINT2006.

Program	# Loops	# LOVs	% with LOVs
400.perlbench	2151	251	12%
401.bzip2	324	80	25%
403.gcc	7344	1816	25%
429.mcf	57	8	14%
445.gobmk	1444	1090	75%
456.hmmer	425	218	51%
458.sjeng	364	140	38%
462.libquantum	78	60	77%
464.h264ref	1526	1192	78%
471.omnetpp	913	280	31%
473.astar	101	65	64%
483.xalancbmk	8637	1938	22%
total	23364	7138	31%

TABLE III
WORST CASE MEMORY OVERHEAD OF EPID TECHNIQUES.

Program	PCCE	SIC	SEI	STAT	LEI	PEPID
400.perlbench	197KiB	8B	59.8MiB	0	110KiB	262KiB
401.bzip2	8B	8B	238MiB	0	5KiB	112B
403.gcc	165KiB	8B	885MiB	0	1.1MiB	496KiB
429.mcf	232B	8B	626MiB	0	5.3KiB	488B
445.gobmk	2.7KiB	8B	16.3MiB	0	126KiB	5.4KiB
456.hmmer	32B	8B	255KiB	0	6.1KiB	96B
458.sjeng	368B	8B	21.3KiB	0	20.4KiB	856B
462.libquantum	8B	8B	40MiB	0	5.2KiB	48B
464.h264ref	24B	8B	121KiB	0	9.7KiB	168B
471.omnetpp	1.9KiB	8B	>7GiB	0	22.4KiB	1.9KiB
473.astar	8B	8B	1.3MiB	0	5.6KiB	64B
483.xalancbmk	246KiB	8B	2.86GiB	0	12.6MiB	431KiB
mean	51.1KiB	8B	436MiB	0	1.2MiB	99.9KiB

and computed the median and 95% confidence interval for the mean. We ran all experiments on a 64-bit Intel i5 machine with 8GB RAM running Ubuntu 13.04. Fig. 10 presents the normalized median of each technique compared to uninstrumented trials of the benchmark suite. We also present the geometric means of the normalized results for each technique. Error bars indicate the 95% confidence intervals of the means.

PCCE and SIC usually have the lowest overhead on average, 8% and 9% respectively. The next closest is PEPID with 25%, then LEI with 70% and SEI with 314%. We immediately see that in comparison to the other inter-execution technique, SEI, PEPID consistently produces lower overhead. The original SEI paper produced overhead near 42% on average, which differs the results we find. While we used `clang`, SEI used `Diablo/FIT` with link time optimization [37], yielding optimization differences. The original evaluation of SEI also used SPEC CPU95 and CPU2000 benchmarks with smaller workloads than those present in the 2006 benchmarks. When we used the 'test' workload, the smallest that SPEC provides, SEI improved to 90% overhead. This illustrates that scalability was indeed a problem for SEI. One of the benchmarks, 471.omnetpp, would not even run using SEI on the reference workload because the stack used for EPIDs consumed all memory and crashed the program before completion. In contrast, PEPID's overhead was always closer to SIC and PCCE, in spite of the fact that it provides a more informative form of EPID.

We also note that the original PCCE paper reports overhead closer to 3%. The work used profile guided instrumentation to achieve low runtime overhead, but we did not use profile guided instrumentation in our LLVM based implementation. Also, while we used `clang` to compile programs, PCCE used `gcc`, which optimizes programs differently. This does not affect our comparison because *all* techniques in this paper were compiled using `clang`. In addition, using profile guided optimizations for PCCE would just strengthen the results of PEPID, since PEPID relies on PCCE as a subtask.

B. Space Overhead

Maintaining the current EPID consumes memory for each technique except STAT. Table III lists the maximum memory overhead for each benchmark and technique as well as the mean across all benchmarks. SIC and STAT require a single word or no overhead, respectively, which may be preferable if memory must be conserved. Even though PCCE compactly encodes the calling context, it still takes 51.1KiB on average because some benchmarks have deeply nested calls. For instance, 403.gcc has a maximum depth of 21100 calls. Profile guided instrumentation can help reduce this. However, even the worst case overhead of PEPID, which uses PCCE, is relatively low, around 100 KiB on average. It is almost always smaller than LEI and is orders of magnitude smaller than SEI in spite of its precision. This makes PEPID a preferable technique for analyses needing inter-execution EPIDs.

C. Client Impact

We now show how a comprehensive technique like PEPID is preferred over a non-comprehensive technique like SEI for a particular dynamic analysis. We consider an analysis known as dual slicing. Dual slicing is a backward slicing technique that contrasts two executions [6]. Instead of including all backward dependences for a slice criterion, it includes only those dependences that either (1) exist in only one of the executions or (2) exist in both executions but define different values. In this way, dual slicing produces explanations for why two executions differ, which can be useful for debugging

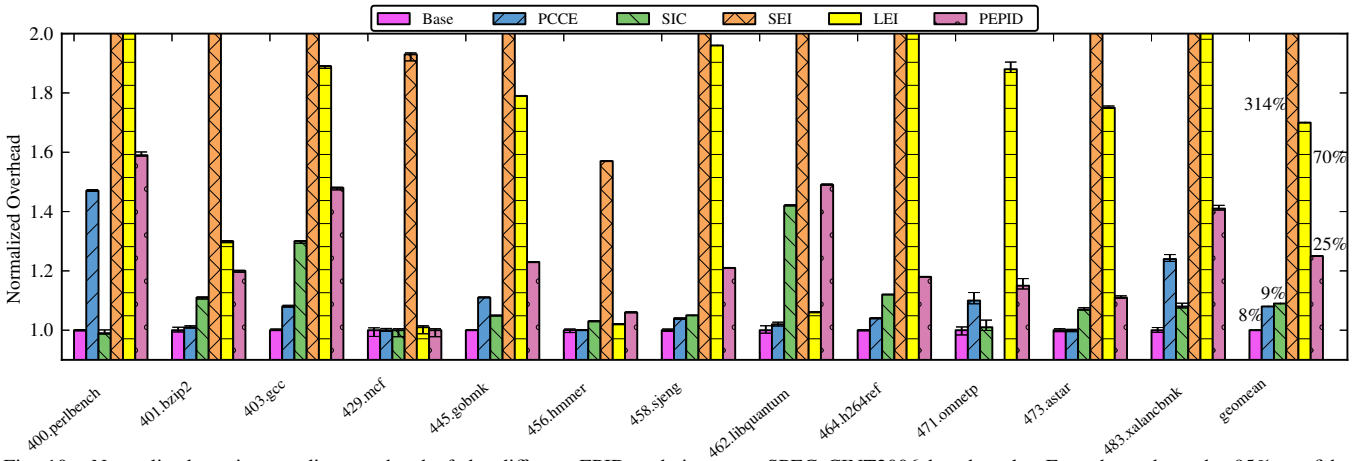

Fig. 10. Normalized runtime median overhead of the different EPID techniques on SPEC CINT2006 benchmarks. Error bars show the 95% confidence intervals for the mean of each technique.

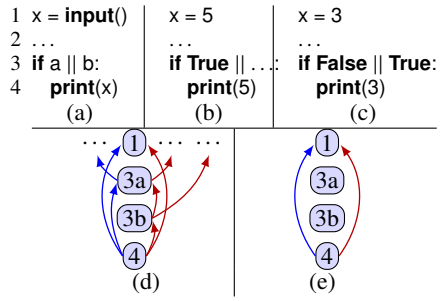

Fig. 11. (a) A program that can lead to bad dual slices using SEI. (b) A trace where a is **True**. (c) A trace where b is **True**. (d) A dual slice using SEI. (e) A dual slice using PEPID.

[6] or for security analysis [12]. Backward slicing techniques traditionally include too many dependences to be practical [38], so dual slicing is particularly useful because it prunes away irrelevant dependences as it contrasts two executions.

EPID techniques like SEI form the foundation of dual slicing. EPIDs determine whether a dependence in one execution exists in another. Unfortunately, when noncomprehensive EPIDs are used, they can include unnecessary dependences in the slice, defeating one of the main goals of the technique.

Consider the program in Fig. 11a. This program reads an integer x from the user and prints it if either a or b is **True**. Suppose there are two different executions of the program, one where the program prints 5, and the other prints 3 as shown in Fig. 11b-c. Note that a is **True** in one execution, but only b is **True** in the other. This matches the case we considered earlier in Section II-C, meaning that the print statements in the two executions have different EPIDs under SEI. Because the EPIDs differ, dual slicing considers them different statements and also includes their control dependences. The dual slice includes the different values of a and b via control dependence, even though they do not actually affect the output differences. These irrelevant dependences get in the way and impede the user's ability to understand why the executions printed different numbers as shown in Fig. 11d. Here, the arrows denote dependences in the dual slice. In contrast, a comprehensive technique like PEPID is able to identify that the print statements occur at the same execution point and identify that the differing user input for x caused the different output. Fig. 11e shows the dual slice when using PEPID and clearly identifies how the input difference directly caused the output difference.

VI. Related Work

We examined several approaches from literature that compute EPIDs for dynamic analyses [13]–[15], [24], [30]. Each of these techniques has been used to solve real problems in dynamic analysis ranging from informing replay techniques [8] to fine-grained execution comparison [6]. The comparison of these techniques along with our new EPID computation technique, PEPID, is one of the core contributions of this work.

In developing PEPID, we based our system around the notion that the position within an unwound and unrolled CFG provides a notion of identity for execution points. This was inspired in part by bounded model checking [33], but model checkers do not need to consider the alternative high-level semantics for unrolling degenerate irreducible loops. Similar notions of identifying execution points also exist within static analysis, where k-CFA provides a statically bounded approximation of execution points using a similar intuition [39].

VII. Conclusion

In this paper, we examined several techniques for computing execution point IDs (EPIDs) and considered their strengths, weaknesses, and limitations. To address limitations of inter-execution EPIDs, we introduced a new technique, PEPID, that is able to comprehensively compute inter-execution EPIDs with significantly less space and runtime overhead than existing techniques. PEPID also produces more meaningful relationships between EPIDs in different executions. Finally, we show that establishing these meaningful relationships is useful in the context of real world dynamic analyses.

Acknowledgments

This research is supported in part by the National Science Foundation (NSF) under grants 0917007, 0845870, and 1218993. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] W. N. Sumner and X. Zhang, "Comparative causality: Explaining the differences between executions," in *ICSE*, 2013.

[2] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "Synergy: a new algorithm for property checking," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14, 2006, pp. 117–127.

[3] G. J. Holzmann, "Economics of software verification," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '01, 2001, pp. 80–89.

[4] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05, 2005, pp. 2–2.

[5] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: a fast address sanity checker," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, ser. USENIX ATC'12, 2012, pp. 28–28.

[6] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan, "Analyzing concurrency bugs using dual slicing," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10, 2010, pp. 253–264.

[7] J.-D. Choi and S. L. Min, "Race frontier: reproducing data races in parallel-program debugging," in *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '91, 1991, pp. 145–154.

[8] M. Ronsse, K. De Bosschere, M. Christiaens, J. C. de Kergommeaux, and D. Kranzlmüller, "Record/replay for nondeterministic program executions," *Commun. ACM*, vol. 46, no. 9, pp. 62–67, Sep. 2003.

[9] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: parallelizing sequential logging and replay," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XVI, 2011, pp. 15–26.

[10] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05, 2005, pp. 1–1.

[11] W. N. Sumner and X. Zhang, "Algorithms for automatically computing the causal paths of failures," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE '09, 2009, pp. 355–369.

[12] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential slicing: Identifying causal execution differences for security applications," in *IEEE Symposium on Security and Privacy*, 2011, pp. 347–362.

[13] B. Xin, W. N. Sumner, and X. Zhang, "Efficient program execution indexing," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '08, 2008, pp. 238–248.

[14] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, "Scalable temporal order analysis for large scale debugging," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009, pp. 44:1–44:11.

[15] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise calling context encoding," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1160–1177, Sep. 2012.

[16] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07, 2007, pp. 89–100.

[17] S. Park, S. Lu, and Y. Zhou, "CTrigger: exposing atomicity violation bugs from their hiding places," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIV, 2009, pp. 25–36.

[18] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, ser. SIGSOFT '02/FSE-10, 2002, pp. 1–10.

[19] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi, "Accurate, efficient, and adaptive calling context profiling," in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '06, 2006, pp. 263–271.

[20] M. D. Bond and K. S. McKinley, "Probabilistic calling context," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, ser. OOPSLA '07, 2007, pp. 97–112.

[21] M. D. Bond, G. Z. Baker, and S. Z. Guyer, "Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10, 2010, pp. 13–24.

[22] T. Mytkowicz, D. Coughlin, and A. Diwan, "Inferred call path profiling," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, ser. OOPSLA '09, 2009, pp. 175–190.

[23] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 29, 1996, pp. 46–57.

[24] J. M. Mellor-Crummey and T. J. LeBlanc, "A software instruction counter," in *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS III, 1989, pp. 78–86.

[25] H. Yu and Z. Li, "Fast loop-level data dependence profiling," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12, 2012, pp. 37–46.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05, 2005, pp. 190–200.

[27] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[28] L. Guo, A. Roychoudhury, and T. Wang, "Accurately choosing execution runs for software fault localization," in *Proceedings of the 15th international conference on Compiler Construction*, ser. CC'06, 2006, pp. 80–95.

[29] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09, 2009, pp. 47–58.

[30] P. Joshi, M. Naik, K. Sen, and D. Gay, "An effective dynamic analysis for detecting generalized deadlocks," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE '10, 2010, pp. 327–336.

[31] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05, 2005, pp. 342–351.

[32] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '99, 1999, pp. 193–207.

[33] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, vol. 2988. Springer, 2004, pp. 168–176.

[34] B. Steensgaard, "Sequentializing program dependence graphs for irreducible programs," Microsoft Research, Redmond, Wash, Tech. Rep. MSR-TR-93-14.

[35] G. Ramalingam, "On loops, dominators, and dominance frontiers," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 5, pp. 455–490, 2002.

[36] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[37] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "Diablo: a reliable, retargetable and extensible link-time rewriting framework," in *2005 IEEE INTERNATIONAL SYMPOSIUM ON SIGNAL PROCESSING AND INFORMATION TECHNOLOGY (ISSPIT), VOLS 1 AND 2*. IEEE, 2005, pp. 7–12.

[38] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '06, 2006, pp. 169–180.

[39] O. Shivers, "Control-flow analysis of higher-order languages," Ph.D. dissertation, Carnegie Mellon University, May 1991.