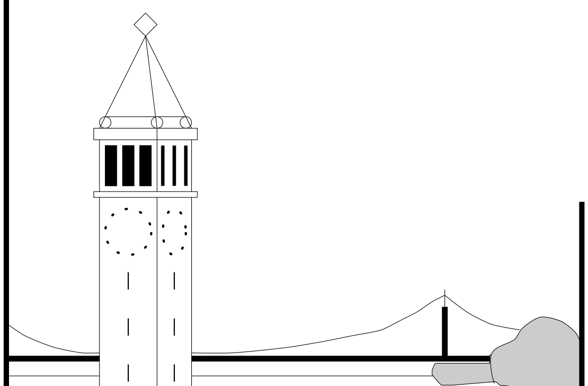
Precise Interprocedural Analysis using Random Interpretation
(Revised version*)

Sumit Gulwani
gulwani@cs.berkeley.edu

George C. Necula
necula@cs.berkeley.edu



Report No. UCB/CSD-4-1353

November 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Precise Interprocedural Analysis using Random Interpretation

(Revised version*)

Sumit Gulwani
gulwani@cs.berkeley.edu

George C. Necula
necula@cs.berkeley.edu

Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720-1776

ABSTRACT

We describe a unified framework for random interpretation that generalizes previous randomized intra-procedural analyses, and also extends naturally to efficient inter-procedural analyses. There is no such natural extension known for deterministic algorithms. We present a general technique for extending any intra-procedural random interpreter to perform a context-sensitive inter-procedural analysis with only polynomial increase in running time. This technique involves computing *random* summaries of procedures, which are complete and probabilistically sound.

As an instantiation of this general technique, we obtain the first polynomial-time randomized algorithm that discovers all linear equalities inter-procedurally in a program that has been abstracted using linear assignments. We also obtain the first polynomial-time randomized algorithm for precise inter-procedural value numbering over a program that uses unary uninterpreted functions.

We present experimental evidence that quantifies the precision and relative speed of the analysis for discovering linear equalities along two dimensions: intra-procedural vs. inter-procedural, and deterministic vs. randomized. We also present results that show the variation of the error probability in the randomized analysis with changes in algorithm parameters. These results suggest that the error probability is much lower than our conservative theoretical bounds.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

This research was supported by NSF Grants CCR-0326577, CCR-0081588, CCR-0085949, CCR-00225610, CCR-0234689, NASA Grant NNA04CI57A, Microsoft Research Fellowship for the first author, and Sloan Fellowship for the second author. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

* An original version of this paper appeared in POPL ’05, January 12-14, 2005, Long Beach, California, USA.

General Terms

Algorithms, Theory, Verification

Keywords

Inter-procedural Analysis, Random Interpretation, Randomized Algorithm, Linear Relationships, Uninterpreted Functions, Inter-procedural Value Numbering

1. INTRODUCTION

A sound and complete program analysis is undecidable [13]. A simple alternative is *random testing*, which is complete but unsound, in the sense that it cannot prove absence of bugs. At the other extreme, we have sound *abstract interpretations* [3], wherein we pay a price for the hardness of program analysis in terms of having an incomplete (i.e., conservative) analysis, or by having algorithms that are complicated and have long running-time. *Random interpretation* is a probabilistically sound program analysis technique that can be simpler, more efficient and more complete than its deterministic counterparts, at the price of degrading soundness from absolute certainty to guarantee with arbitrarily high probability [8, 9].

Until now, random interpretation has been applied only to intra-procedural analysis. Precise inter-procedural analysis is provably harder than intra-procedural analysis [16]. There is no general recipe for constructing a precise and efficient inter-procedural analysis from just the corresponding intra-procedural analysis. The functional approach proposed by Sharir and Pnueli [22] is limited to finite lattices of dataflow facts. Sagiv, Reps and Horwitz have generalized the Sharir-Pnueli framework to build context-sensitive analyses, using graph reachability [17], even for some kind of infinite domains. They successfully applied their technique to detect linear constants inter-procedurally [20]. However, their generalized framework requires appropriate distributive transfer functions as input. There seems to be no obvious way to automatically construct context-sensitive transfer functions from just the corresponding intra-procedural analysis. We show in this paper that if the analysis is based on random interpretation, then there is a general procedure for lifting it to perform a precise and efficient inter-procedural analysis.

Our technique is based on the standard procedure summarization approach to inter-procedural analysis. However, we compute randomized procedure summaries that are probabilistically sound. We show that such summaries can be computed efficiently, and we prove that the error probability, which is over the random choices made by the algorithm, can be made as small as desired by controlling various parameters of the algorithm.

We instantiate our general technique to two abstractions, linear arithmetic (Section 8) and unary uninterpreted functions (Section 9), for which there exist intra-procedural random interpretation based analyses. For the case of linear arithmetic, our technique yields a more efficient algorithm than the existing algorithms for solving the same problem. For the case of unary uninterpreted functions, we obtain the first polynomial-time and precise algorithm that performs inter-procedural value numbering [1] over a program with unary uninterpreted function symbols.

In the process of describing the inter-procedural randomized algorithms, we develop a generic framework for describing both intra-procedural and inter-procedural randomized analyses. This framework generalizes previously published random interpreters [8, 9], guides the development of randomized interpreters for new domains, and provides a large part of the analysis of the resulting algorithms. As a novel feature, the framework emphasizes the discovery of relationships, as opposed to their verification, and provides generic probabilistic soundness results for this problem.

Unlike previous presentations of random interpretation, we discuss in this paper our experience with implementing and using an inter-procedural random interpreter on a number of C programs. In Section 10, we show that the error probability of such algorithms is much lower in practice than predicted by the theoretical analysis. This suggests that tighter probability bounds may be obtained. We also compare experimentally the randomized inter-procedural analysis for discovering linear equalities with an intra-procedural version [8] and with a deterministic algorithm [20] for the related but simpler problem of detecting constant variables.

This paper is organized as follows. In Section 2, we present a generic framework for describing intra-procedural randomized analyses. In Section 3, we explain the two main ideas behind our general technique of computing random procedure summaries. In Section 4, we formally describe our generic algorithm for performing an inter-procedural randomized analysis. We prove the correctness of this algorithm in Section 5. We discuss fixed-point computation in Section 6, and the computational complexity of the algorithm in Section 7. We instantiate this generic algorithm to obtain an inter-procedural algorithm for discovering linear relationships, and for value numbering in Section 8 and Section 9 respectively. Section 10 describes our experiments.

2. INTRA-PROCEDURAL RANDOM INTERPRETATION

In this section, we formalize the intra-procedural random interpretation technique, using a novel framework that generalizes existing random interpreters.

2.1 Preliminaries

We first describe our program model. We assume that the flowchart representation of a program consists of nodes of the kind shown in Figure 1(a), (b), (c), and (d). In the assignment node, x refers to a program variable while e denotes some expression. A non-deterministic assignment $x :=?$ denotes that the variable x can be assigned any value. Such non-deterministic assignments are used as a safe abstraction of statements (in the original source program) that our abstraction cannot handle precisely. Non-deterministic conditionals, represented by $*$, denote that the control can flow to either branch irrespective of the program state before the conditional. They are used as a safe abstraction of conditional guards that our abstraction cannot handle precisely.

An execution of a random interpreter over a program computes a state ρ at each program point π. A state is a mapping from program variables (that are visible at the corresponding program point) to values v that are polynomials over the finite field $\mathbb{F}_p$ for some prime p. ($\mathbb{F}_p$ denotes the field of integers $\{0, \ldots, p\text{-}1\}$ where arithmetic is done modulo p.) These polynomials may simply be elements of $\mathbb{F}_p$ (as in the case of linear arithmetic [8]), vectors of elements from ${\mathbb{F}_p}^1$ (as in the case of uninterpreted functions [9]), or linear functions of program's input variables (as we will see for the case of inter-procedural analysis). The notation $\rho[x \leftarrow v]$ denotes the state obtained from ρ by setting the value of variable x to value v.

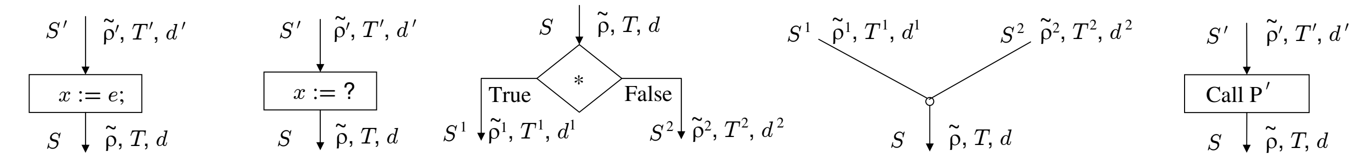
Often a random interpreter performs multiple, say t executions of a program in parallel, the result of which is a sequence of t states at each program point. We refer to such a sequence of states as a *sample* S and we write S_i to refer to the ith state in sample S.

A random interpreter processes ordinary assignments $x := e$ by setting the value of variable x to the value of expression e in the state before the assignment node. Expressions are evaluated using an `Eval` function, which depends on the underlying domain of the analysis. We give some examples of `Eval` functions in Section 2.3, where we also describe the properties that an `Eval` function must satisfy. We use the notation $\texttt{Eval}(e, \rho)$ to denote the value of expression in state ρ as computed by the random interpreter.

Non-deterministic assignments $x :=?$ are processed by assigning a fresh random value to variable x. A random interpreter executes both branches of a conditional. At join points, it performs a random affine combination of the joining states using the affine join operator ϕ_w. We describe this operator and its properties in Section 2.2. In presence of loops, a random interpreter goes around loops until a fixed point is reached. It follows from Theorem 5 that the number of steps required to reach a fixed point is bounded above by $1 + k_{\mathrm{v}}$, where k_{v} is the maximum number of program variables visible at any program point. However, we do not know if there is an efficient way to detect a fixed point since the random interpreter works with randomized data-structures. The random interpreter can use the strategy of iterating around a loop (maximal strongly connected component) $(1+k_{\mathrm{v}})\beta$ times, where β is the number of back-edges in the loop. Note that this guarantees that a fixed point will be reached.

In Section 2.4, we discuss how to verify or discover expression equivalences from such random executions of a program. We also give a bound on the error probability in this process.

[1] Note that a vector $(v_1, \ldots, v_\ell)$ can be represented by the polynomial $\sum_{i=1}^{\ell} z_i v_i$, where $z_1, \ldots, z_\ell$ are some fresh variables.



(a) Assignment Node (b) Non-deterministic Assignment Node (c) Non-deterministic Conditional Node (d) Join Node (e) Procedure Call

Figure 1: Flowchart nodes

Finally, in Section 2.5, we give an example of the random interpretation technique for linear arithmetic to verify assertions in a program.

We use the notation $\Pr(E)$ to denote the probability of event E over the random choices made by a random interpreter. Whenever the interpreter chooses some random value, it does so independently of the previous choices and uniformly at random from $\mathbb{F}_p$.

2.2 Affine Join Operator

The affine join operator ϕ_w takes as input two values v_1 and v_2 and returns their affine join with respect to the weight w as follows:

$$\phi_w(v_1, v_2) \stackrel{def}{=} w \times v_1 + (1 - w) \times v_2$$

The affine join operator can be thought of as a selector between v_1 and v_2, similar to the ϕ functions used in static single assignment (SSA) form [4]. If $w = 1$ then $\phi_w(v_1, v_2)$ evaluates to v_1, and if $w = 0$ then $\phi_w(v_1, v_2)$ evaluates to v_2. The power of the ϕ_w operator comes from the fact that a non-boolean (random) choice for w captures the effect of both the values v_1 and v_2.

The affine join operator can be extended to states ρ in which case the affine join is performed with the same weight for each variable. Let ρ_1 and ρ_2 be two states, and x be a program variable. Then,

$$\phi_w(\rho_1, \rho_2)(x) \stackrel{def}{=} \phi_w(\rho_1(x), \rho_2(x))$$

For any polynomial Q and any state ρ, we use the notation $[\![Q]\!]\rho$ to denote the result of evaluation of polynomial Q in state ρ. The affine join operator has the following useful properties. Let Q and Q' be two polynomials that are linear over program variables (and possibly non-linear over other variables). Let $\rho_w = \phi_w(\rho_1, \rho_2)$ for some randomly chosen w from a set of size p. Then,

A1. Completeness: If Q and Q' are equivalent in state ρ_1 as well as in state ρ_2, then they are also equivalent in state ρ_w.

$$([\![Q]\!]\rho_1 = [\![Q']\!]\rho_1) \wedge ([\![Q]\!]\rho_2 = [\![Q']\!]\rho_2) \Rightarrow [\![Q]\!]\rho_w = [\![Q']\!]\rho_w$$

A2. Soundness: If Q and Q' are not equivalent in either state ρ_1 or state ρ_2, then it is unlikely that they will be equivalent in state ρ_w.

$$([\![Q]\!]\rho_1 \neq [\![Q']\!]\rho_1) \vee ([\![Q]\!]\rho_2 \neq [\![Q']\!]\rho_2) \Rightarrow \Pr([\![Q]\!]\rho_w = [\![Q']\!]\rho_w) \leq \frac{1}{p}$$

2.3 SEval Function

The random interpreter processes an assignment node $x := e$ by updating the value of variable x to the value of expression e in the state before the assignment node. Expressions are evaluated using the `Eval` function, which depends on the underlying abstract domain of the analysis. The `Eval` function takes an expression e and a state ρ and computes some value v. The `Eval` function plays the same role as an abstract interpreter's transfer function for an assignment node. `Eval` is defined in terms of a symbolic function `SEval` that translates an expression into a polynomial over the field $\mathbb{F}_p$. This polynomial is linear in program variables, and may contain random variables as well, which stand for random field values chosen during the analysis. (The `SEval` function for linear arithmetic has no random variables, while the `SEval` function for uninterpreted functions uses random variables.) `Eval`(e, ρ) is computed by replacing program variables in `SEval`(e) with their values in state ρ, replacing the random variables with the random values that have chosen for them, and then evaluating the result over the field $\mathbb{F}_p$. (The random values r_j are chosen for the random variables y_j once for each execution of the program, and in each execution the same value r_j is used for all occurrences of the random variable y_j.) Following are examples of two `Eval` functions that have been described in previous papers.

`SEval` *function for Linear Arithmetic.* The random interpretation for linear arithmetic is described in a previous paper [8]. The following language describes the expressions in this abstract domain. Here x refers to a variable and c refers to an arithmetic constant.

$$e ::= x \quad | \quad e_1 \pm e_2 \quad | \quad c \times e$$

The `SEval` function for this abstraction simply translates the linear arithmetic operations to the corresponding field operations. In essence, `Eval` simply evaluates the linear expression over the field $\mathbb{F}_p$.

$$\texttt{SEval}(e) = e$$

`SEval` *function for Unary Uninterpreted Functions.* The random interpretation for uninterpreted functions is described in a previous paper [9]. We show here a simpler `SEval` function, for the case of unary uninterpreted functions. The following language describes the expressions in this abstract domain. Here x refers to a variable and F refers to a unary uninterpreted function.

$$e ::= x \quad | \quad F(e)$$

The SEval function for this abstraction is as follows.

$$\begin{aligned}\texttt{SEval}(x) &= x \\ \texttt{SEval}(F(e)) &= r_1 \times \texttt{SEval}(e) + r_2\end{aligned}$$

Here r_1 and r_2 refer to random variables, unique for each unary uninterpreted function F. Note that in this case, SEval produces polynomials that have degree more than 1, although still linear in the program variables.

The SEval function corresponding to the Eval function for binary uninterpreted functions (as described in [9]) evaluates expressions to vectors (of polynomials). Note that a vector $(v_1, \ldots, v_\ell)$ can however be represented as the polynomial $z_1 v_1 + \ldots + z_\ell v_\ell$ where $z_1, \ldots, z_\ell$ are fresh variables that do not occur in the program.

2.3.1 Properties of SEval Function

The SEval function should have the following properties. Let x be any variable and e_1 and e_2 be any expressions. Then,

B1. Soundness: The SEval function should not introduce any new equivalences.

$$\texttt{SEval}(e_1) = \texttt{SEval}(e_2) \quad \Rightarrow \quad e_1 = e_2$$

Note that the first equality is over polynomials, while the second equality is in the analysis domain.

B2. Completeness: The SEval function should preserve all equivalences.

$$e_1 = e_2 \quad \Rightarrow \quad \texttt{SEval}(e_1) = \texttt{SEval}(e_2)$$

B3. Referential transparency:

$$\texttt{SEval}(e_1[e_2/x]) \quad = \quad \texttt{SEval}(e_1)[\texttt{SEval}(e_2)/x]$$

This property (along with properties B1 and B2) is needed to prove the correctness of the action of the random interpreter for an assignment node $x := e$. As mentioned earlier, the random interpreter computes the state after an assignment node by updating the value of x to the value of the polynomial $\texttt{SEval}(e)$ in the state before the assignment node.

B4. Linearity: The SEval function should be a polynomial that is linear in program variables. This property is needed to prove the completeness of the random interpreter across join nodes, where it uses the affine join operator to merge program states.

Properties B1 and B3 are necessary for proving the probabilistic soundness of the random interpreter. Property B2 or property B4 need not be satisfied if completeness is not an issue. This may happen when the underlying abstraction is difficult to reason about, yet one is interested in a (probabilistically) sound and partially complete reasoning for that abstraction. For example, the following SEval function for "bitwise or operator" ($||$) satisfies all the above properties except property B2.

$$\texttt{SEval}(e_1 || e_2) \quad = \quad \texttt{SEval}(e_1) + \texttt{SEval}(e_2)$$

This SEval function models commutativity and associativity of the $||$ operator. However, it does not model the fact that $x || x = x$. In this paper, we assume that the SEval function satisfies all the properties mentioned above. However, the results in this paper can also be extended to prove relative completeness for of the random interpreter if the SEval function does not satisfy property B2 or B4.

Also, note that the SEval function for linear arithmetic as described above has the soundness property for linear arithmetic over the prime field $\mathbb{F}_p$. The problem of reasoning about linear arithmetic over rationals can be reduced to reasoning about linear arithmetic over $\mathbb{F}_p$, where p is chosen randomly. This is a randomized reduction with some error probability, and is discussed further in Section 8.

2.4 Error Probability Analysis

A random interpreter performs multiple executions of a program as described above, the result of which is a collection of t states, or a sample, at each program point. We can use these samples to verify and discover equivalences.

The process of verifying equivalences and the bound on the error probability can be stated for a generic random interpreter. The process of discovering equivalences is abstraction specific; however a generic error probability bound can be stated for this process when the SEval function does not involve any random variables (e.g., SEval function for linear arithmetic).

2.4.1 Verifying Equivalences

Let S be the sample computed by the random interpreter at some program point π after fixed-point computation. The random interpreter declares two expressions e_1 and e_2 to be equivalent at program point π iff $\texttt{Eval}(e_1, S_i) = \texttt{Eval}(e_2, S_i)$ for all states S_i in the sample S. We denote this by $S \models e_1 = e_2$. Two expressions e_1 and e_2 are equivalent at program point π iff for all program states ρ that may arise at program point π in any execution of the program, $[\![e_1]\!]\rho = [\![e_2]\!]\rho$. We denote this by $\pi \models e_1 = e_2$.

The following properties hold:

C1. The random interpreter verifies all valid equivalences.

$$\pi \models e_1 = e_2 \quad \Rightarrow \quad S \models e_1 = e_2$$

C2. With high probability, any equivalence verified by the random interpreter is correct.

$$\begin{aligned}\pi \not\models e_1 = e_2 \quad \Rightarrow \quad \Pr(S \models e_1 = e_2) &\leq \left(\frac{d}{p}\right)^t \\ d \quad = \quad (n_{\texttt{jp}} + \delta)(1 + k_{\texttt{v}})\beta\end{aligned}$$

Here $n_{\texttt{jp}}$ denotes the maximum number of join points in any procedure, and $k_{\texttt{v}}$ denotes the maximum number of program variables visible at any program point. δ refers to the maximum degree of $\texttt{SEval}(e)$ for any expression that uses as many function symbols as there are in any procedure. For example, δ is 1 for the linear arithmetic abstraction. For the abstraction of uninterpreted functions, δ is bounded above by the number of function symbols that occur in any procedure. Intuitively, we add $n_{\texttt{jp}}$ to δ to account for the multiplications with the random weights at join points. Note that the error probability in property C2 can be made arbitrarily small by choosing p to be greater than d and choosing t appropriately.

We now briefly sketch the proofs of properties C1 and C2. Consider a symbolic version of the random interpreter (similar to the one introduced in Appendix A for the case of inter-procedural analysis) that computes a symbolic state at each program point, i.e., a state in which variables are mapped to polynomials in terms of the random weight variables corresponding to join points, and any other random variables used in the `SEval` function. The fully symbolic random interpreter discovers exactly the set of valid equivalences at all program points. This can be proved by induction on the number of flowchart nodes analyzed by the random interpreter (similar to the proof of completeness and soundness of the fully-symbolic random interpreter for inter-procedural analysis given in Appendix A). Note that each state in any sample computed by the random interpreter is an independent random instantiation of the corresponding symbolic state computed by the fully-symbolic random interpreter. Property C1 now follows directly. Property C2 follows from the error bound for the Schwartz and Zippel's polynomial identity testing algorithm [21, 23], which states that the probability that two distinct polynomials (of degree at most d) evaluate to same values on t independent random instantiations (from a set of size p) is bounded above by $(\frac{d}{p})^t$.

2.4.2 *Discovering Equivalences*

We now move our attention to the issue of discovering equivalences. Although, this usage mode was apparent from previous presentations of random interpretation, we have not considered until now the probability of erroneous judgment for this case.

The process of discovering equivalences at a program point is abstraction specific. For example, Section 8.1 describes how to discover equivalences for the abstraction of linear arithmetic.

The following property states an upper bound on the probability that the random interpreter discovers any false equivalence at a given program point for the case when `SEval` function does not involve any random variables. Let S be the sample computed by the random interpreter at some program point π. Then, the following holds:

C3. With high probability, all equivalences discovered by the random interpreter are valid.

$$\Pr(\exists e_1, e_2 : \pi \not\models e_1 = e_2 \ \wedge \ S \models e_1 = e_2) \leq \frac{n}{1-\alpha}\alpha^{t-k_{\mathrm{v}}}$$

where $\alpha = \frac{3dt}{p(t-k_{\mathrm{v}})}$, $d = n_{\mathtt{jp}}(1 + k_{\mathrm{v}})\beta$.

Here n refers to the total number of program points. The proof of property C3 follows from a more general property (Property D3 in Section 5.2) that we prove for analyzing the correctness of an inter-procedural random interpreter. Note that we must choose t to be at least $1 + k_{\mathrm{v}}$, and p greater than $3dt$.

This completes the description of the generic framework for random interpretation. Next, we show an example, and then we proceed to extend the framework to describe inter-procedural analyses.

2.5 Example

In this section, we illustrate the random interpretation scheme for discovering linear equalities among program variables by means of an example.

Consider the procedure shown in Figure 2 (ignoring for the moment the states shown on the side). We first consider

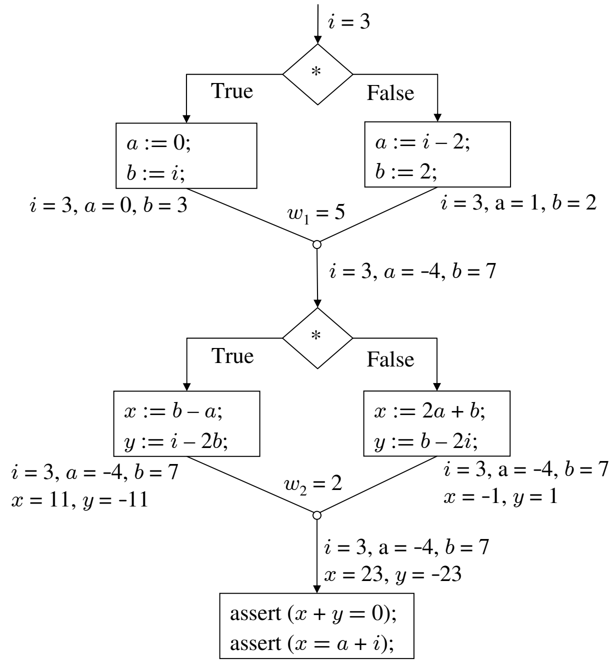


Figure 2: A code fragment with four paths. Of the two equations asserted at the end the first one holds on all paths but the second one holds only on three paths. The values of variables shown next to each edge represent the program states computed in a random interpretation.

this procedure in isolation of the places it is used (i.e., intra-procedural analysis). Of the two assertions at the end of the procedure, the first is true on all four paths, and the second is true only on three of them (it is false when the first conditional is false and the second is true). Regular testing would have to exercise that precise path on which the second assertion fails, to avoid inferring that the second equality holds. Random interpretation is able to invalidate the second assertion in just one (non-standard) execution of the procedure.

The random interpreter starts with a random value 3 for the input variable i and then executes the assignment statements on both sides of the conditional using the `Eval` function for linear arithmetic, which matches with the standard interpretation of linear arithmetic. In the example, we show the values of all live variables at each program point. The two program states before the first join point are combined with the affine join operator using the random weight $w_1 = 5$. Note that the resulting program state after the first join point can never arise in any real execution of the program. However, this state captures the invariant that $a + b = i$, which is necessary to prove the first assertion in the procedure. The random interpreter then executes both sides of the second conditional and computes an affine join of the two states before the second join point using the random weight $w_2 = 2$. We can easily verify that the resulting state at the end of the procedure satisfies the first assertion but does not satisfy the second. Thus, in one run of the procedure we have noticed that one of the (potentially) exponentially many paths breaks the second assertion. Note

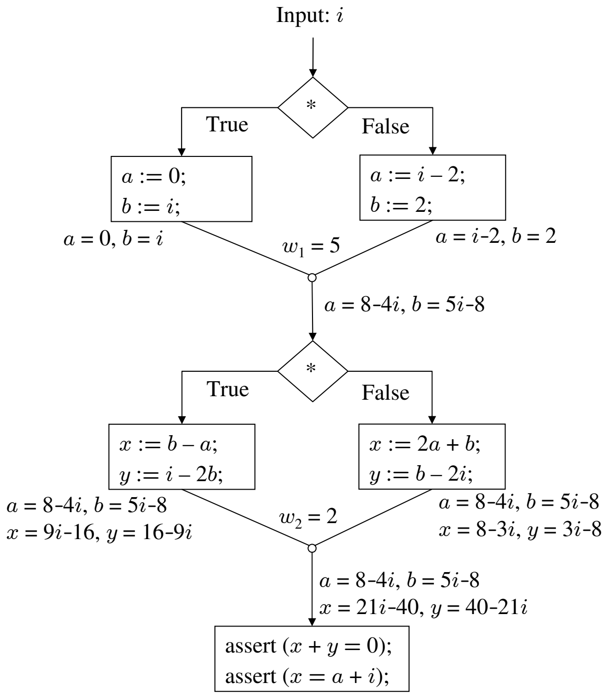


Figure 3: Illustration of random symbolic interpretation on the program shown in Figure 2. Note that the second assertion is true in the context $i = 2$, and the random symbolic program state at the end of the program satisfies it in that context.

that choosing w to be either 0 or 1 at a join point corresponds to executing either the true branch or the false branch of its corresponding conditional; this is what naive testing accomplishes. However, by choosing w (randomly) from a set that also contains non-boolean values, we are able to capture the effect of both branches of a conditional in just one interpretation of the program. In fact, there is a very small chance that the random interpreter will choose such values for i, w_1 and w_2 that will make it conclude that both assertions hold (e.g., $i = 2$, or $w_1 = 1$).

In an inter-procedural setting, the situation is more complicated. If this procedure is called only with input argument $i = 2$, then both assertions hold, and the analysis is expected to infer that. One can also check that if the random interpreter chooses $i = 2$, then it is able to verify both the assertions, for any choice of w_1 and w_2. We look next at what changes are necessary to extend the analysis to be inter-procedural.

3. KEY IDEAS

Inter-procedural random interpretation is based on the standard summary-based approach to inter-procedural analysis. Procedure summaries are computed in the first phase, and actual results are computed in the second phase. The real challenge is in computing context-sensitive summaries, i.e., summaries that can be instantiated with any context to yield the most precise behavior of the procedures under that context. A context for a procedure refers to any relevant information regarding the values that the input variables of

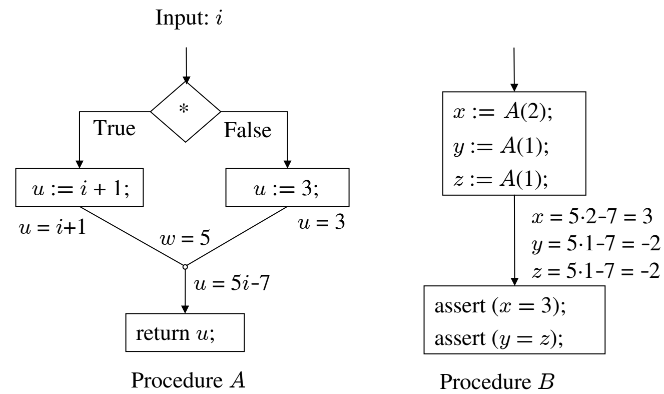


Figure 4: A program that demonstrates unsoundness of a single random symbolic run. Note that the first assertion at the end of procedure B is true, while the second assertion is not true since procedure A may take different branches in different runs.

that procedure can take.

In this section, we briefly explain the two main ideas behind the summary computation technique that can be used to perform a precise inter-procedural analysis using the `SEval` function of a precise intra-procedural random interpreter.

3.1 Random Symbolic Run

Intra-procedural random interpretation involves interpreting a program using random values for the input variables. The state at the end of the procedure can be used as a summary for that procedure. However, such a summary will not be context-sensitive. For example, consider the procedure shown in Figure 2. The second assertion at the end of the procedure is true in the context $i = 2$, but this conditional fact is not captured by the random state at the end of the procedure.

Observe that in order to make the random interpretation scheme context-sensitive, we can simply delay choosing random values for the input variables. Instead of using states that map variables to field values, we use states that map variables to linear functions of input variables. This allows the flexibility to replace the input variables later depending on the context. However, we continue to choose random weights at join points and perform a random affine join operation.

As an example, consider again the procedure from before, shown now in Figure 3. Note that the random symbolic state at the end of the procedure (correctly) does not satisfy the second assertion. However, in a context where $i = 2$, the state does satisfy $x = a + i$ since x evaluates to 2 and a to 0. This scheme of computing partly symbolic summaries is surprisingly effective and guarantees context-sensitivity, i.e., it entails all valid equivalences in all contexts.

3.2 Multiple Runs

Consider the program shown in Figure 4. The first assertion in procedure B is true. However, the second assertion is false because the non-deterministic conditional in procedure A can branch differently in the two calls to procedure A,

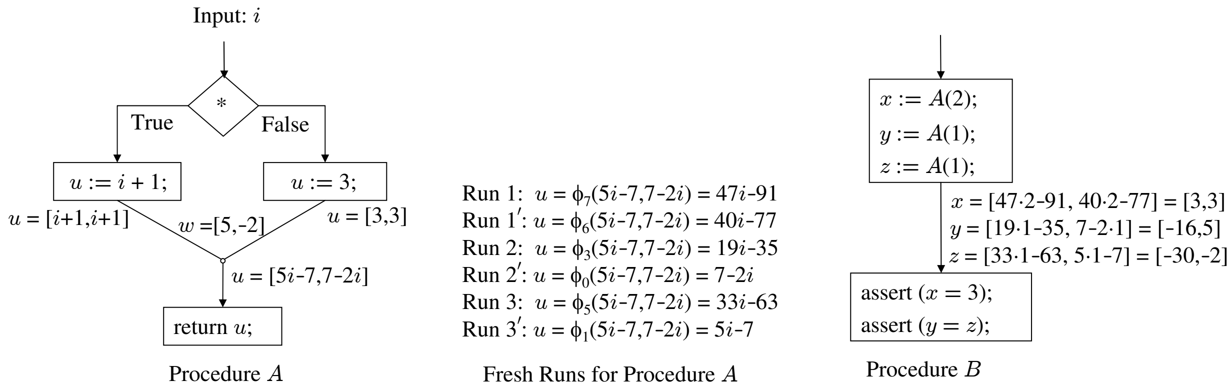


Figure 5: Illustration of multiple random symbolic runs for inter-procedural analysis on the program also shown in Figure 4. In this example, 2 random symbolic runs are computed for each procedure, and are further used to generate a fresh random symbolic run for every call to that procedure. Run j and Run j' are used at the j^{th} call site of procedure A while computing the two runs for procedure B. Note that this technique is able to correctly validate the first assertion and falsify the second one.

even with the same input. If we use the same random symbolic run for procedure A at different call sites in procedure B, then we incorrectly conclude that the second assertion holds. This happens because use of the same run at different call sites assumes that the non-deterministic conditionals in the called procedure are resolved in the same manner in different calls. This problem can be avoided if a fresh or independent run is used at each call point. By *fresh run*, we mean a run computed with a fresh choice of random weights at the join points.

One approach to generate a fresh run for each call site is to compute summaries that are parametrized by weight variables (i.e., instead of using random weights for performing the affine join operation, we use symbolic weight variables). Then, for each call site, we can instantiate this summary with a fresh set of random weights for the weight variables. The problem with this approach is that the symbolic coefficients of linear functions of input variables, which are assigned to procedure variables (in states computed by the random interpreter) may have an exponential-size representation.

Another approach to generate m fresh runs for any procedure P is to execute m times the random interpretation scheme for procedure P, each time with a fresh set of random weights. However, this may require computing an exponential number of runs for other procedures. For example, consider a program in which each procedure P_i calls procedure P_{i+1} two times. To generate a run for P_0, we need 2 fresh runs for P_1, which are obtained using 4 fresh runs for P_2, and so on.

The approach that we use is to generate the equivalent of t fresh runs for any procedure P from t fresh runs of each of the procedures that P calls (for some parameter t that depends on the underlying abstraction). This approach relies on the fact that a random affine combination (i.e., a random weighted combination with sum of the weights being 1) of t runs of a procedure yields the equivalent of a fresh run for that procedure. For an informal geometric intuition, note that we can obtain any number of fresh points in a 2-dimensional plane by taking independent random affine combinations of three points that span the plane.

In Figure 5, we revisit the program shown in Figure 4 and illustrate this random interpretation technique of using a fresh run of a procedure at each call site. Note that we have chosen $t = 2$. The t runs of the procedure are shown in parallel by assigning a tuple of t values to each variable in the program. Note that procedure B calls procedure A three times. Hence, to compute 2 fresh runs for procedure B, we need to generate 6 fresh runs for procedure A. The figure shows generation of 6 fresh runs (Runs 1,1′,2,2′,3, and 3′) from the 2 runs for procedure A. The first call to procedure A uses the first two (Runs 1 and 1′) of these 6 runs, and so on. Note that the resulting program states at the end of procedure B satisfy the first assertion, but not the second assertion thereby correctly invalidating it.

4. INTER-PROCEDURAL RANDOM INTERPRETATION

We now describe the precise inter-procedural random interpretation for any abstraction that is equipped with an `SEval` function that has the properties discussed in Section 2.3.

4.1 Notation

We first describe our program model. A program is a set of procedures, each with one entry and one exit node. We assume that each procedure has been abstracted using the flowchart nodes shown in Figure 1. A procedure call node is simply denoted by the name of the procedure P' that is being called. For simplicity, we assume that the inputs and outputs of a procedure being called are passed as global variables. We express the computational complexity of algorithms in terms of the number of assignment nodes and for that purpose, we assume that the expression e in an assignment node is of constant size. We use the following notation:

- n : Number of nodes.
- $n_{\mathtt{a}}$: Number of assignment nodes.

- $n_{\mathtt{j}}$: Number of join nodes.
- $n_{\mathtt{p}}$: Number of procedure call nodes.
- $n_{\mathtt{jp}}$: Maximum number of join nodes in any procedure.
- $n_{\mathtt{pp}}$: Maximum number of procedure call nodes in any procedure.
- $k_{\mathtt{v}}$: Maximum number of program variables visible at any program point.
- $k_{\mathtt{i}}$: Maximum number of input variables for any procedure.
- $k_{\mathtt{o}}$: Maximum number of output variables for any procedure.

The set of input variables of a procedure P includes the set of all global variables read by procedure P directly as well as the set of input variables for any procedure P' called by P. Similarly for the set of output variables of a procedure P.

The random interpreter uses an optimization (described in Section 4.3) that involves converting the program into SSA form [4]. With regard to that, we use the following notation:

- $n_{\mathtt{s}}$: Number of total assignment statements (both phi assignments and non-phi assignments) in SSA version of the program.

It has been reported [4] that the ratio of the number of new phi-assignments introduced (as a result of SSA conversion) to the number of original assignments typically varies between 0.3 to 2.8 (i.e., $1.3n_{\mathtt{a}} \leq n_{\mathtt{s}} \leq 3.8n_{\mathtt{a}}$) irrespective of program size.

For describing a bound on the number of steps required for fixed-point computation, we use the following notation:

- β : Maximum number of back-edges in any program loop

For a structured flow-graph, β denotes the maximum loop nesting depth. Based on the experiments that we carried out, we noticed β to be a small constant in practice, usually bounded above by 3.

4.2 Basic Algorithm

The inter-procedural random interpreter performs a standard two-phase computation. The first phase, or the bottom-up phase, computes procedure summaries by starting with leaf procedures. The second phase, or top-down phase, computes the actual results of the analysis at each program point by using the summaries computed in the first phase. In presence of loops in the call graph and inside procedures, both phases require fixed-point computation, which we address in Section 6.

The random interpreter starts by choosing random elements $r_j \in \mathbb{F}_p$ for any random variables y_j that are used in the `SEval` function. Every occurrence of variable y_j is replaced by the same random choice r_j in all runs[2] (unlike intra-procedural case, in which fresh random values are chosen for each run) when evaluating any expression using the `Eval` function. We use the notation $\mathtt{SEval}'(e)$ to denote the polynomial obtained from $\mathtt{SEval}(e)$ by replacing all occurrences of the random variables y_j by the random elements r_j that have been chosen (globally) for them. The prime p is chosen to ensure that the error probability of the random interpreter (which is a function of p among other parameters, as described in Theorem 3) is small. A 32-bit prime is usually sufficient in practice. We now describe the two-phase computation performed by the random interpreter.

4.2.1 *Phase 1*

A summary for a procedure P, denoted by Y_P, is either $\perp$ (denoting that the procedure has not yet been analyzed, or on all paths it transitively calls procedures that have not yet been analyzed), or is a collection of t runs $\{Y_{P,i}\}_{i=1}^{t}$. A run of procedure P is a mapping from output variables of procedure P to random symbolic values, which are linear expressions in terms of the input variables of procedure P. The number of runs t should be greater than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$ for probabilistic soundness, as predicted by our theoretical estimates. However, experiments (discussed in Section 10) suggest that a smaller value of t does not yield any error in practice.

To compute a procedure summary, the random interpreter computes a sample S at each program point, as shown in Figure 1. A sample is either $\perp$ or a sequence of t states. A state at a program point π is a mapping of program variables (visible at point π) to random symbolic linear expressions in terms of the input variables of the enclosing procedure. We use the notation S_i to denote the i^{th} state in sample S. The random interpreter computes a sample S at each program point from the samples at the immediately preceding program points, and using the summaries computed so far for the called procedures. The transfer functions for the flowchart nodes are described below. After the random interpreter is done interpreting a procedure, it computes the summary of that procedure by simply projecting the sample (or the t states) at the end of the procedure to the output variables of the procedure.

Initialization. The random interpreter starts by initializing the summaries of all procedures, and the samples at all program points except at procedure entry points to $\perp$. The samples at procedure entry points are initialized by setting all input variables x to $\mathtt{SEval}'(x)$ in all states.

$$S_i(x) = \mathtt{SEval}'(x)$$

Note that $\mathtt{SEval}'(x)$ is simply x for the abstractions of linear arithmetic and uninterpreted functions.

Assignment Node. See Figure 1 (a).
If the sample S' before the assignment node is $\perp$, then the sample S after the assignment node is defined to be $\perp$. Otherwise, the random interpreter computes S by updating the value of variable x in each state of sample S' as follows.

$$S_i = S'_i[x \leftarrow \mathtt{Eval}(e, S'_i)]$$

Non-deterministic Assignment Node. See Figure 1 (b).
If the sample S' before the non-deterministic assignment node is $\perp$, then the sample S after the non-deterministic

[2] This is required to prove completeness of the inter-procedural random interpreter for the operation of producing a fresh run of a procedure by an affine combination of the runs in its summary.

assignment node is defined to be $\perp$. Else, the random interpreter processes the assignment $x :=?$ by transforming each state in the sample S' by setting x to some fresh random value.

$$S_i = S'_i[x \leftarrow v], \text{ where } v = \texttt{SEval}'(y)[\texttt{Rand}()/y]$$

The fresh random value v is obtained from the polynomial $\texttt{SEval}'(y)$ by substituting variable y by a randomly chosen element from $\mathbb{F}_p$.

Non-deterministic Conditional Node. See Figure 1 (c). The random interpreter simply copies the sample S before the conditional node on the two branches of the conditional.

$$S^1 = S \text{ and } S^2 = S$$

Join Node. See Figure 1 (d). If any one of the samples S^1 or S^2 before the join node is $\perp$, the random interpreter assigns the other sample before the join node to the sample S after the join node. Otherwise, the random interpreter selects t random weights $w_1, \ldots, w_t$ and computes the affine join of S^1 and S^2 with respect to those weights to obtain the sample S after the join node.

$$S = \phi_{[w_1,\ldots,w_t]}(S^1, S^2)$$

Procedure Call. See Figure 1 (e). If the sample S' before the procedure call is $\perp$, or if the summary $Y_{P'}$ is $\perp$, then the sample S after the procedure call is defined to be $\perp$. Otherwise the random interpreter executes the procedure call as follows. The random interpreter first generates t fresh random runs $Y_1, \ldots, Y_t$ for procedure P' using the current summary (t runs) for procedure P'. Each fresh run Y_i for procedure P' is generated by taking a random affine combination of the t runs in the summary of procedure P'. This involves choosing random weights $w_{i,1}, \ldots, w_{i,t}$ with the constraint that $w_{i,1} + \cdots + w_{i,t} = 1$, and then doing the following computation. Then,

$$Y_i(x) = \sum_{j=1}^{t} w_{i,j} \times Y_{P',j}(x)$$

The effect of a call to procedure P' is to update the values of the variables that are written to by procedure P'. The random interpreter models this effect by updating the values of these variables using the fresh random runs Y_i (computed above) as follows. Let the input (global) variables of procedure P' be $y_1, \ldots, y_k$. Let $O_{P'}$ denote the set of output (global) variables of procedure P'.

$$S_i(x) = \begin{cases} Y_i(x)[S'_i(y_1)/y_1, \ldots, S'_i(y_k)/y_k] & \text{if } x \in O_{P'} \\ S'_i(x) & \text{otherwise} \end{cases}$$

4.2.2 Phase 2

For the second phase, the random interpreter also maintains a sample S (which is a sequence of t states) at each program point, as in phase 1. However, unlike phase 1, the states in phase 2 map variables to values that do not involve input variables. The samples are computed for each program point from the samples at the preceding program points in the same manner as in phase 1 except for the initialization, which is done as follows:

Initialization. The random interpreter initializes the samples at all program points except at procedure entry points to $\perp$. The sample at the entry point of the `Main` procedure is initialized by setting all input variables x to fresh random values in all states.

$$S_i(x) = \texttt{SEval}'(x)[\texttt{Rand}()/x]$$

As before a fresh random value is obtained from the polynomial $\texttt{SEval}'(x)$ by substituting variable x by a randomly chosen element from $\mathbb{F}_p$.

The sample S at the entry point of any other procedure P is obtained as a random affine combination of all the non-$\perp$ samples at the call sites to P. Let these samples be $S^1, \ldots, S^k$. Then for any input variable x,

$$S_i(x) = \sum_{j=1}^{k} w_{i,j} \times S^j_i(x)$$

where $w_{i,1}, \ldots, w_{i,k}$ are random weights with the constraint that $w_{i,1} + \cdots + w_{i,k} = 1$, for all $1 \le i \le t$. This affine combination encodes all the relationships (among input variables of procedure P) that hold in all calls to procedure P.

4.3 Optimization

Maintaining a sample explicitly at each program point is expensive (in terms of time and space complexity) and redundant. For example, consider the samples before and after an assignment node $x := e$. They differ (at most) only in the values of variable x.

An efficient way to represent samples at each program point is to convert all procedures into minimal SSA form [4] and to maintain one global sample for each procedure instead of maintaining a sample at each program point. The values of a variable x in the sample at a program point π are represented by the values of the variable $v_{x,\pi}$ in the global sample, where $v_{x,\pi}$ is the renamed version of variable x at program point π after the SSA conversion. Under such a representation, interpreting an assignment node or a procedure call simply involves updating the values of the modified variables in the global sample. Interpreting a join node involves updating the values of ϕ variables at that join point in the global sample.

This completes the description of the inter-procedural random interpretation. Next, we estimate the error probability of the random interpreter.

5. ERROR PROBABILITY ANALYSIS

In this section, we estimate the error probability of the random interpreter. We show that the random interpreter is complete, i.e. it validates all correct equivalences (property D4). On the other hand, we show that with high probability (over the random choices made by the random interpreter), the random interpreter does not validate a given incorrect equivalence (Theorem 3). We also show that if the `SEval` function does not involve any random variables (e.g., `SEval` function for linear arithmetic), then with high probability (over the random choices made by the random interpreter), the random interpreter validates only correct equivalences (Theorem 3). For the purpose of establishing these results, we first state and prove some useful properties of the samples computed by the random interpreter in phase 1 and phase 2.

5.1 Analysis of Phase 1

We first introduce some terminology. We use the term *input context*, or simply *context*, for a procedure P to denote a mapping of input variables of procedure P to polynomials that are linear in program variables. For any context C, let $Abs(C)$ denote the set of equivalences (involving variables that have mappings in C) in the abstract domain that are implied by C, i.e., $Abs(C) = \{e_1 = e_2 \mid \texttt{Eval}(e_1, C) = \texttt{Eval}(e_2, C)\}$. For any polynomial Q, and any context (or state) C, we use the notation $Q[C]$ to denote the polynomial obtained from Q by substituting all variables that have mappings in C by those mappings.

Let π be some program point in procedure P. Let A be some set of paths that lead to π from the entry point of procedure P. We say that an equivalence $e_1 = e_2$ holds at π along paths A in context C iff the weakest precondition of the equivalence $e_1 = e_2$ along all paths in A belongs to the set $Abs(C)$. We denote this by $Holds(e_1 = e_2, A, C)$.

We say that a state ρ entails an equivalence $e_1 = e_2$ in context C, denoted by $\rho \models_C e_1 = e_2$, when $\texttt{Eval}(e_1, \rho)[C] = \texttt{Eval}(e_2, \rho)[C]$. We say that a sample S entails an equivalence $e_1 = e_2$ in context C, denoted by $S \models_C e_1 = e_2$, when all states in S do so.

Let S be the sample computed by the random interpreter (in phase 1) at a program point π in procedure P after analyzing a set of paths A. The following properties hold.

D1. Soundness (Phase 1): Suppose that the `SEval` function does not involve any random variables. With high probability, in all input contexts, S entails only the equivalences that hold at π along the paths analyzed by the random interpreter (i.e., with high probability, for all input contexts C and all equivalences $e_1 = e_2$, $\neg(Holds(e_1 = e_2, A, C)) \Rightarrow \neg(S \models_C e_1 = e_2)$). The error probability $\gamma_1(S)$ (assuming that the samples computed before computation of S satisfy property D1) is bounded above as follows:

$$\gamma_1(S) \quad \leq \quad p^{k_{\mathrm{i}}}\left(\frac{\alpha^{t-k_{\mathrm{v}}}}{1-\alpha}\right), \text{ where } \alpha = \frac{3 d_S t}{p(t-k_{\mathrm{v}})}$$

We use the notation d_S to refer to the number of join points and procedure calls along any path analyzed by the random interpreter immediately after computation of sample S. A formal definition of d_S is given in Appendix A.

D2. Completeness (Phase 1): In all input contexts, S entails all equivalences that hold at π along the paths analyzed by the random interpreter (i.e., for all input contexts C and all equivalences $e_1 = e_2$, $Holds(e_1 = e_2, A, C) \Rightarrow S \models_C e_1 = e_2$).

For the purpose of proving property D1, we hypothetically extend the random interpreter to compute a *fully-symbolic state* at each program point, i.e., a state in which variables are mapped to polynomials in terms of the input variables and random weight variables corresponding to join points and procedure calls (see Appendix A for details). A key part of the proof strategy is to prove that the fully-symbolic state at each point captures *exactly* the set of equivalences at that point in any context along the paths analyzed by the random interpreter. In essence, a fully-symbolic interpreter is sound and complete, even though it might be computationally expensive. The proof of this fact is by induction on the number of flowchart nodes analyzed by the random interpreter (Lemma 9 in Appendix A). We now prove property D1 using the following two steps.

We first bound the error probability that a sample S with t states does not entail exactly the same set of equivalences as the corresponding fully-symbolic state $\tilde{\rho}$ in *a given context*. The following lemma specifies a bound on this error probability, which we denote by $\gamma_1'(S)$.

LEMMA 1. *$\gamma_1'(S) \leq \frac{\alpha^{t-k_{\mathrm{v}}}}{1-\alpha}$, where $\alpha = \frac{3d_S t}{p(t-k_{\mathrm{v}})}$.*

The proof of Lemma 1 is in Appendix C.

Next we observe that it is sufficient to analyze the soundness of a sample in a smaller number of contexts (compared to the total number of all possible contexts), which we refer to as a basic set of contexts. If a sample entails exactly the same set of equivalences as the corresponding fully-symbolic state for all contexts in a basic set, then it has the same property for all contexts. Let N denote the number of contexts in any smallest basic set of contexts. The following theorem specifies a bound on N.

LEMMA 2. *$N \leq p^{k_{\mathrm{i}}}$.*

The proof of Lemma 2 is in Appendix D.

The probability that a sample S is not sound in any of the contexts is bounded above by the probability that S is not sound in some given context multiplied by the size of any basic set of contexts. Thus, the error probability $\gamma_1(S)$ mentioned in property D1 is bounded above by $\gamma_1'(S) \times N$.

The proof of property D2 is by induction on the number of flowchart nodes analyzed by the random interpreter, and is similar to the proof of completeness of the fully symbolic state given in Appendix A.

5.2 Analysis of Phase 2

A sample S computed by the random interpreter in phase 2 at a program point π in procedure P has the following properties.

D3. Soundness (Phase 2): Suppose that the `SEval` function does not involve any random variables. With high probability, S entails only the equivalences that hold at π along the paths analyzed by the random interpreter. The error probability $\gamma_2(S)$ (assuming that the samples computed before computation of sample S satisfy property D3, and all samples computed in phase 1 satisfy property D1) is bounded above as follows:

$$\gamma_2(S) \quad \leq \quad \frac{\alpha^{t-k_{\mathrm{v}}}}{1-\alpha}, \text{ where } \alpha = \frac{3 d_S t}{p(t-k_{\mathrm{v}})}$$

d_S is as described in property D1. A formal definition of d_S is given in Appendix A.

D4. Completeness (Phase 2): S entails all equivalences that hold at π along the paths analyzed by the random interpreter.

The proof of property D3 is an instantiation of the proof of Lemma 1, and follows from it by choosing the context C to be the identity mapping. The proof of property D4 is an instantiation of the proof of property D2, and follows from it by choosing the context C to be the identity mapping.

Property D4 implies that the random interpreter discovers all valid equivalences. We now use the properties D1 and D3 to prove the following theorem, which establishes a bound on the total error probability of the random interpreter.

Theorem 3 (Probabilistic Soundness Theorem). *Let $H_1 = 1 + k_{\mathtt{v}}(k_{\mathtt{i}} + 1)$ and $H_2 = 1 + k_{\mathtt{v}}$. Let $q = nH_1\beta$ and $d = max\{(n_{\mathtt{jp}} + n_{\mathtt{pp}})H_1\beta, (n_{\mathtt{j}} + n_{\mathtt{p}})H_2\beta\}$. Suppose that $p > (3dt)^2$. If* `SEval` *function does not involve any random variables, then the probability that all random samples computed by the random interpreter satisfy only those equivalences that hold at the corresponding program points is at least $1 - \frac{2q}{1-\alpha}\alpha^{t-t_0}$, where $\alpha = \frac{3dt}{p(t-k_{\mathtt{v}})}$ and $t_0 = k_{\mathtt{v}} + 2k_{\mathtt{i}}$. In general, the probability that the random interpreter does not verify a given false equivalence $e_0 = e_0'$ is bounded below by $1 - \frac{2q}{1-\alpha}\alpha^{t-t_0} - \frac{\delta}{p}$. Here δ refers to the maximum degree of* `SEval`(e) *for any expression e that uses a maximum of $2sH_2(nH_1)^{nH_1}(nH_2)^{nH_2} + s'$ function symbols, where s is the maximum number of function symbols in any assignment node, and s' is the maximum number of function symbols in expressions e_0 and e_0'.*

Proof. It follows from the discussion after Theorem 4 and Theorem 5 in the next section that the random interpreter goes around each loop at most $H_1\beta$ times in phase 1 and $H_2\beta$ times in phase 2 for fixed-point computation. Hence, the random interpreter computes at most $nH_1\beta$ samples in phase 1 and $nH_2\beta$ samples in phase 2. Also, note that the value of d_S in the bounds on the probabilities $\gamma_1(S)$ and $\gamma_2(S)$ (which bound the unsoundness of a sample S) is at most $(n_{\mathtt{jp}}+n_{\mathtt{pp}})H_1\beta$ in phase 1 and $(n_{\mathtt{j}}+n_{\mathtt{p}})H_2\beta$ in phase 2. This implies that the total error probability of the random interpreter for the case when `SEval` function does not involve any random variables is bounded above by $\frac{2q}{1-\alpha}\alpha^{t-t_0}$.

We now prove an upper bound on the probability that the random interpreter with a general `SEval` function validates an incorrect equivalence. Let P_0 be the original program, and let $e_0 = e_0'$ be some equivalence that does not hold in program P_0 at some program point π. Let P_1 be the program obtained from P_0 by replacing all expressions e by `SEval`(e). Let $e_1 =$ `SEval`(e_0) and $e_1' =$ `SEval`(e_0'). It follows from the properties B1, B2 and B3 of the `SEval` function that the equivalence $e_1 = e_1'$ does not hold in program P_1 (at program point π).

Let P_2 be the program obtained from P_1 by substituting all random variables y_j in the `SEval` function by the random values r_j chosen by the random interpreter. Similarly, let $e_2 = e_1[r_j/y_j]$ and $e_2' = e_1'[r_j/y_j]$. We now show that the probability (over the choice of the random values r_j) that P_2 does not satisfy $e_2 = e_2'$ (at program point π) is bounded above by $\frac{\delta}{p}$. Let P_1' and P_2' be the programs without any procedure calls obtained from programs P_1 and P_2 respectively using the procedure inlining technique described in Appendix B. The programs P_1' and P_2' satisfy the same set of equivalences as the programs P_1 and P_2 respectively at the corresponding points. Each procedure in the programs P_1' and P_2' has at most $n_{max} = 2(nH_1)^{nH_1}(nH_2)^{nH_2}$ nodes. Since P_1' does not satisfy the equivalence $e_1 = e_1'$ (at program point π) it follows from Theorem 5 that there must be a path of length $n_{max}H_2$ (from the entry point of the procedure enclosing point π to π) along which the equivalence $e_1 = e_1'$ is not satisfied. Let ρ_1 be the state obtained by executing the program P_1' along that path. Note that $e_1[\rho_1] \neq e_1'[\rho_1]$. Let $\rho_2 = \rho_1[r_j/y_j]$. The degrees of the polynomials $e_2[\rho_2]$ and $e_2'[\rho_2]$ are bounded above by δ. It follows from the Schwartz and Zippel's polynomial identity testing theorem that the probability that $e_2[\rho_2] = e_2'[\rho_2]$ is at most $\frac{\delta}{p}$. Hence, the probability that P_2' (or equivalently P_2) satisfies the equivalence $e_2 = e_2'$ (at program point π) is at most $\frac{\delta}{p}$.

Now suppose that the program P_2 does not satisfy the equivalence $e_2 = e_2'$ (at program point π). Observe that performing random interpretation over program P_0 (using the `SEval` function for the underlying abstraction) to decide the validity of the equivalence $e_0 = e_0'$ (at program point π) is equivalent to performing random interpretation over program P_2 (using the identity `SEval` function) to decide the validity of the equivalence $e_2 = e_2'$ (at program point π). It follows from the result established for the case when `SEval` function does not involve any random variables that the probability that the random interpreter validates the incorrect equivalence $e_2 = e_2'$ in program P_2 (at program point π) is at most $\frac{2q}{1-\alpha}\alpha^{t-t_0}$. The desired result now follows from the union bound on this probability and the probability that program P_2 satisfies the equivalence $e_2 = e_2'$ (at program point π). □

Note that for the case when `SEval` function does not involve any random variables, Theorem 3 gives a bound on the error probability for the process of discovering (all) equivalences; while for the general case, it specifies a bound on the error probability for verification of a (single) given equivalence[3]. The theorem implies that for probabilistic soundness we need to choose t to be greater than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$. It may be possible to prove better bounds on t for specific abstractions (e.g., we only require $t > 6$ for the case of unary uninterpreted functions, as discussed in Section 9). For the case when `SEval` function does not involve any random variables, p needs to be greater than $(3dt)^2$. However, for the general case, we need to choose p to be greater than δ, which implies that we need to perform arithmetic with numbers that require $O(\log \delta)$ bits for representation. For example, the value of δ for the theory of unary uninterpreted functions is $2sH_2(nH_1)^{nH_1}(nH_2)^{nH_2} + s'$, and this implies that the arithmetic should be performed with numbers that require $O(nk_{\mathtt{v}} \log n)$ bits for representation. However, we feel that this analysis is very conservative, and experiments suggest that 32-bit primes are good enough in practice.

6. FIXED-POINT COMPUTATION

The notion of loop that we consider for fixed-point computation is that of a maximal strongly connected component (SCC). For defining SCCs in a program in an interprocedural setting, we consider the directed graph representation of a program that has been referred to as supergraph in the literature [17]. This directed graph representation consists of a collection of flowcharts, one for each procedure in the program, with the addition of some new edges. For every edge to a call node, say from node m_1 to call node m_2 with the call being to procedure P, we add two new edges: one from node m_1 to start node of procedure P, and the other from exit node of procedure P to node m_2. Now consider the DAG of SCCs of this directed graph representation of the program. Note that an SCC in this DAG may contain nodes of more than one procedure[4] (in which case

[3]The error probability for verification of m equivalences can be obtained by multiplying the error probability of one equivalence by m.

[4]This happens when the call graph of the program contains a maximal strongly connected component of more than one node.

it contains all nodes of those procedures).

In both phase 1 and phase 2, the random interpreter processes all SCCs in the DAG in a top-down manner. It goes around each SCC until a fixed point is reached. In phase 1, a sample computed by the random interpreter represents sets of equivalences, one for each context. A fixed point is reached for an SCC in phase 1, if for all points π in the SCC and for all contexts C (for the procedure enclosing point π), the set of equivalences at π in context C has stabilized. In phase 2, a sample computed by the random interpreter represents a set of equivalences; and a fixed point is reached for an SCC, if for all points π in the SCC, the set of equivalences at π has stabilized. Let H_1 and H_2 be the upper bounds on the number of iterations required to reach a fixed point across any SCC in phase 1 and 2 respectively.

Theorem 4 (Fixed Point Theorem for Phase 1). *Let $\tilde{\rho}^1, \ldots, \tilde{\rho}^m$ be the fully-symbolic states computed by the random interpreter in phase 1 at some point π inside a loop in successive iterations of that loop such that $\tilde{\rho}^i$ does not imply the same set of equivalences as $\tilde{\rho}^{i+1}$ in some context. Then, $m \leq H_1$, where $H_1 = 1 + k_{\mathtt{v}}(k_{\mathtt{i}} + 1)$.*

Proof. We first introduce some notation. Let ρ be any state that maps variables $x_1, \ldots, x_a$ to polynomials that are linear in input variables $y_1, \ldots, y_b$. Let $\rho(x_j) = v_{j(b+1)} + \sum_{i=1}^{b} v_{i+(j-1)(b+1)} y_i$, where the polynomials v_i do not involve the input variables $y_1, \ldots, y_b$. We use the notation $R(\rho)$ to denote the vector $(v_1, \ldots, v_{a(b+1)}, 1)$. For any fully-symbolic state $\tilde{\rho}$, we use the notation $\mathcal{V}(\tilde{\rho})$ to denote the smallest vector space generated by $\{R(\tilde{\rho})[v_i/w_i] \mid v_i \in \mathbb{F}_p\}$ over the field $\mathbb{F}_p$, where $R(\tilde{\rho})[v_i/x_i]$ denotes the vector obtained from $R(\tilde{\rho})$ by replacing all weight variables w_i by some choices of elements v_i from $\mathbb{F}_p$.

Consider the vector spaces $\mathcal{V}(\tilde{\rho}^i)$ and $\mathcal{V}(\tilde{\rho}^{i+1})$. Note that $\mathcal{V}(\tilde{\rho}^i)$ is included in $\mathcal{V}(\tilde{\rho}^{i+1})$ since $\tilde{\rho}^i$ is an instantiation of $\tilde{\rho}^{i+1}$. Also, note that $\mathcal{V}(\tilde{\rho}^i) \neq \mathcal{V}(\tilde{\rho}^{i+1})$ since $\tilde{\rho}^i$ does not imply the same set of equivalences as $\tilde{\rho}^{i+1}$ in some context. Hence, $Rank(\mathcal{V}(\tilde{\rho}^i)) < Rank(\mathcal{V}(\tilde{\rho}^{i+1}))$. Note that $Rank(\mathcal{V}(\tilde{\rho}^1)) \geq 1$ and $Rank(\mathcal{V}(\tilde{\rho}^m)) \leq 1 + k_{\mathtt{v}}(k_{\mathtt{i}}+1)$. Hence, $m \leq 1 + k_{\mathtt{v}}(k_{\mathtt{i}} + 1)$. □

Theorem 5 (Fixed Point Theorem for Phase 2). *Let $\tilde{\rho}^1, \ldots, \tilde{\rho}^m$ be the fully-symbolic states computed by the random interpreter in phase 2 at some point π inside a loop in successive iterations of that loop such that $\tilde{\rho}^i$ does not imply the same set of equivalences as $\tilde{\rho}^{i+1}$ in some context. Then, $m \leq H_2$, where $H_2 = 1 + k_{\mathtt{v}}$.*

The proof of Theorem 5 is similar to the proof of Theorem 4 and follows from the observation that $Rank(\mathcal{V}(\tilde{\rho}^m)) \leq 1 + k_{\mathtt{v}}$.

We have described a worst-case bound on the number of iterations required to reach a fixed point. However, we do not know if there is an efficient way to detect a fixed point since the random interpreter works with randomized data-structures. The random interpreter can use the strategy of iterating around a loop (SCC) $H_1\beta$ times in phase 1 and for $H_2\beta$ times in phase 2, where β is the number of back-edges in the loop. Note that this guarantees that a fixed point will be reached. This is because if a fixed point is not reached, then the set of equivalences corresponding to at least one of the samples (computed by the random interpreter) at the target of the back-edges must change, and it follows from Theorem 4 and Theorem 5 that there can be at most H_1 or H_2 such changes at each program point in phase 1 or phase 2 respectively.

7. COMPUTATIONAL COMPLEXITY

We assume that the random interpreter performs the optimization of maintaining one global state in the SSA version of the program as discussed in Section 4.3. Under that optimization, a join operation reduces to processing phi-assignments at that join point. We also assume unit cost for each arithmetic operation and that the size of polynomial `SEval`(e) is linear in size of expression e. We estimate the running time of the random interpreter for phase 1, which dominates the running time for phase 2. The cost of processing each assignment, both phi and non-phi, is $O(k_{\mathtt{i}}t)$. The cost of processing a procedure call is $O(k_{\mathtt{i}}k_{\mathtt{o}}t^2)$. For fixed-point computation, the random interpreter goes around each loop at most $H_1\beta$ times. Assuming β to be a constant, the running time of the random interpreter is $O(n_{\mathtt{s}}H_1k_{\mathtt{i}}t + n_{\mathtt{p}}H_1k_{\mathtt{i}}k_{\mathtt{o}}t^2)$. It follows from Theorem 3 that for probabilistic soundness, we need to choose t to be greater than $k_{\mathtt{v}}+2k_{\mathtt{i}}$. This yields a total complexity of $O(n_{\mathtt{s}}k_{\mathtt{v}}^2k_{\mathtt{i}}^2 + n_{\mathtt{p}}k_{\mathtt{v}}^3k_{\mathtt{i}}^2k_{\mathtt{o}})$ for the random interpreter.

If we regard $k_{\mathtt{i}}$ and $k_{\mathtt{o}}$ to be constants, since they denote the size of the interface between procedure boundaries and are supposedly small, and the number of procedure call nodes $n_{\mathtt{p}}$ to be significantly smaller than the number of assignment nodes $n_{\mathtt{s}}$, then the complexity of the random interpreter reduces to $O(n_{\mathtt{s}}k_{\mathtt{v}}^2)$, which is linear in the size of the program and quadratic in the maximum number of visible program variables at any program point.

8. SPECIAL CASE OF LINEAR ARITHMETIC

In this section we discuss the use of inter-procedural random interpretation to discover linear equalities among program variables that take rational values. The basic strategy is to discover the linear equalities among program variables over a randomly chosen prime field $\mathbb{F}_p$ (rather than the infinite field of rationals) using the `SEval` function described in Section 2.3. This is followed by mapping back the discovered equalities to the field of rationals.

The motivation for first solving the problem over a randomly chosen prime field (as opposed to directly solving over the infinite field of rationals) comes from the need to avoid manipulating large numbers. For example, consider the program shown in Figure 6. Any summary-based approach for discovering linear equalities will essentially compute the following summaries for procedures P_m and P'_m:

$$\begin{aligned} P_m(x) &= 2^{2^m}x \\ P'_m(x) &= 2^{2^m}x \end{aligned}$$

Note that representing the arithmetic constant 2^{2^m} using standard decimal notation requires $\Omega(2^m)$ digits. The problem of manipulating large numbers can be avoided by performing computations over a small finite field $\mathbb{F}_p$, where prime p is chosen randomly.

In the next section, we discuss how to discover linear equalities (over the field of rationals) at any program point from the sample computed at that point by performing ran-

```
P_0(x)   = { return 2x; }

P_i(x)   = { y := P_{i-1}(x); return P_{i-1}(y); }

P'_0(x)  = { return 2x; }

P'_i(x)  = { y := P'_{i-1}(x); return P'_{i-1}(y); }

Main() = { y_1 := P_m(0); y_2 := P'_m(0);

           assert(y_1 = y_2); }
```

Figure 6: A program of size $O(m)$ for which any deterministic summary based inter-procedural analysis requires $\Omega(2^m)$ space and time for manipulating arithmetic constants (assuming standard binary representation). The program contains $2m+3$ procedures `Main`, P_i and P'_i for $i \in \{0, \ldots, m\}$. A randomized analysis does not have this problem.

dom interpretation over the field $\mathbb{F}_p$ for some randomly chosen prime p.

8.1 Discovering Linear Equalities

For discovering linear equalities at a program point π, the random interpreter extracts relationships from the sample S computed by it at program point π. This is done by assuming a relationship of the form $\alpha + \sum_j \alpha_j x_j = 0$ among the variables x_j's that are visible at π, and then solving for the unknowns α and α_j's from the following set of simultaneous linear equations:

$$\alpha + \sum_j \alpha_j S_i(x_j) = 0 \qquad 1 \le i \le t$$

The above system of equations may have a parametrized solution (instead of a unique solution). From the parametrized solution, we may obtain a linearly independent set of solutions by repeatedly plugging 1 for one of the parameters and 0 for the rest. A more useful set of linear equalities may be those that involve few variables (as opposed to potentially all visible variables), e.g., variable equalities, constant variables, or induction variables. These may be discovered by assuming more specific templates like $x = \alpha$ or $x = \alpha_1 y + \alpha_2$, and solving for the unknowns α's as above.

Note that the coefficients of the linear equalities thus discovered are expressed modulo p (since the random interpreter performs arithmetic modulo p to avoid the problem of dealing with large numbers). The challenge now is to obtain the original rational coefficients, assuming that these rational coefficients involve integers with small absolute value. In other words, given $z \in \mathbb{F}_p$, we want to obtain small integers m_1 and m_2 such that $\frac{m_1}{m_2} = z \bmod p$, i.e., $m_1 = m_2 z \bmod p$. Note that if both m_1 and m_2 have absolute value less than $\sqrt{\frac{p}{2}}$, then m_1 and m_2 are unique.

Given $z \in \mathbb{F}_p$, the problem of finding smallest m_1 and m_2 such that $\frac{m_1}{m_2} = z \mod p$ is equivalent to the problem of finding the shortest (in terms of the Euclidean norm) non-zero vector in the set $A = \{(m_1, m_2) \mid m_1 - m_2 z = 0 \bmod p\}$. The latter problem can be solved by using *lattice reduction* techniques [2]. Since the set A is 2-dimensional

```
Rationalize(z,p)
    u_1 = (z,1);
    u_2 = (p,0);
    do
        if (||u_1|| > ||u_2||) then swap u_1 and u_2;
        a := ⌊<u_1,u_2> / ||u_1||^2⌉;
        u_2 := u_2 - a u_1;
    while (||u_1|| > ||u_2||);
    return u_1;
```

Figure 7: A procedure to obtain small rational coefficients from their images in $\mathbb{F}_p$. Here $\langle u_1, u_2 \rangle$ denotes the inner product of vectors u_1 and u_2, i.e., if $u_1 = (a_1, b_1)$ and $u_2 = (a_2, b_2)$, then $\langle u_1, u_2 \rangle = a_1 a_2 + b_1 b_2$. $\|u\|$ denotes the Euclidean norm $\sqrt{\langle u, u \rangle}$, and $\lfloor f \rceil$ denotes the integer closest to f.

in this case, we can simply use *Gaussian reduction* algorithm [5], which is similar to the Euclidean algorithm for computing the greatest common divisor of two numbers. The procedure `Rationalize` shown in Figure 7 implements this algorithm and returns the pair m_1 and m_2, given z and p.

The correctness of the procedure `Rationalize` follows from the invariant that both vectors u_1 and u_2 belong to the set A and their norm decreases in each iteration of the while loop. The while loop terminates with u_1 being the shortest vector in set A and u_2 being the next shortest vector. We now sketch a proof that the while loop in the procedure `Rationalize` terminates in at most $\lceil \log 2p \rceil$ iterations. The value of $\|u_2\|$ in the first iteration is p. Its value in the last iteration is at least $\sqrt{\frac{p}{2}}$, which is a lower bound for the length of the second shortest vector in the set A. (This is because there can be at most one pair of m_1 and m_2 such that $\frac{m_1}{m_2} = z \mod p$ and $|m_1|, |m_2| < \sqrt{\frac{p}{2}}$. Hence, it must be the case that at least one element of either the shortest vector or the second shortest vector must have absolute value at least $\sqrt{\frac{p}{2}}$. This implies that the length of the second shortest vector is bounded below by $\sqrt{\frac{p}{2}}$.) It can be proved that $\|u_2\|$ decreases by a factor of at least $\sqrt{2}$ in each iteration. Hence, the loop terminates in at most $\lceil \log_{\sqrt{2}} \left(\frac{p}{\sqrt{\frac{p}{2}}} \right) \rceil$ iterations. However, experiments show that on the average 6 iterations are needed for a 32-bit prime.

We now describe some heuristics (as an alternative to implementing the procedure `Rationalize`) to discover the original rational coefficients from their images in $\mathbb{F}_p$. We can compute and store the following mapping I_p (indexed by its image) for a randomly chosen prime p and some small integer constant c beforehand.

$$I_p\left(\frac{m_1}{m_2}\right) = m_1 \times m_2^{-1} \bmod p \qquad -c \le m_1 \le c,\ 1 \le m_2 \le c$$

Hence, given z, we can lookup the store to immediately output m_1 and m_2 such that $\frac{m_1}{m_2} = z \bmod p$. This approach works if the absolute values of the numerator and denominator are at most c.

Another alternative is to assume that the denominators of the coefficients of the linear equalities are 1, or in fact any known constant m. In this case, given z and m, we can estimate m_1 such that $z = \frac{m_1}{m} \bmod p$ as follows:

$$m_1 = \begin{cases} m_1' & \text{if } m_1' < \frac{p}{2} \\ m_1' - p & \text{otherwise} \end{cases}$$

where

$$m_1' = (z \times m) \bmod p$$

In either of the above solutions (for obtaining rational numbers from their images in $\mathbb{F}_p$), we need to verify the linear equalities thus discovered. This is because we do not know beforehand whether the assumption on the numerators and denominators (of the rational coefficients in the linear equalities) being small hold or not. Hence, we need to run the random interpreter again with a new randomly chosen prime p' and verify the linear equalities discovered in the first round.

In the next section, we estimate the additional error probability that arises in the process of mapping the linear equalities discovered over a randomly chosen prime field to the field of rationals. We then discuss the computational complexity of the inter-procedural random interpreter for the abstraction of linear arithmetic.

8.2 Error Probability Analysis

The error probability of the random interpreter can be decomposed into two parts. The special case in Theorem 3 gives the error probability of the random interpreter assuming that the linear equalities are to be discovered over the prime field $\mathbb{F}_p$. We now estimate the remaining error probability that results from performing the computations over the prime field $\mathbb{F}_p$ instead of the infinite field of rationals (Theorem 6), and then mapping the results back to rationals Theorem 8). This error probability is a function of the size of the set from which the prime is chosen randomly; hence, it suggests how big the size of this set should be in order to obtain a specific upper bound on the error probability.

Performing computations over a prime field preserves all true linear equalities, but may introduce some spurious linear equalities. For example, consider the following program fragment, where c is some prime number.

```
if (*) then x := 1 else x := c + 1;
assert (x = 1);
```

The assertion at the end of the program is false, but if the arithmetic is performed over the prime field $\mathbb{F}_p$ for $p = c$, then the assertion becomes true. However, note that the probability of choosing the prime number p to be c is small since the prime number p is chosen from a large enough set of primes. It follows from Theorem 6 (stated and proved below) that, in general, the probability of such spurious linear equalities being introduced is small.

The process of discovering the true coefficients of linear equalities from the coefficients expressed in the prime field can also introduce some error. For example, consider the following program fragment:

$$x := c;$$

The linear equality $x = c$ holds at the end of the above program fragment. Suppose $p \le c \le 3p/2$. Let $c' = c \bmod p$. Then, the technique suggested in Section 8.1 will yield the incorrect linear equality $x = c'$, if $c' \bmod p' = c \bmod p'$, where p' is the second randomly chosen prime for performing random interpretation over the prime field $\mathbb{F}_p'$ to verify the equalities discovered during the first round. However, it follows from Property 7 (stated below) that the probability of choosing the prime number p' to be such that $c' \bmod p' = c \bmod p'$ is small since the prime number p' is chosen from a large enough set of primes. It follows from Theorem 8 (stated and proved below) that in general the probability of discovering such incorrect coefficients in linear equalities is small.

Before stating the theorems that bound the desired error probabilities, we first introduce some notation. Let s be a bound on the size of all expressions e that occur in the assignment node $x := e$ or conditional node $e = 0$ in terms of the number of additions. Let c_m be a bound on the absolute value of the coefficients that occur in these expressions. Let $b_m = 1 + n_{max}(n_{max}+1)(\log s + \log c_m)$ bits, where $n_{max} = 2(nH_1)^{nH_1}(nH_2)^{nH_2}$.

THEOREM 6. *The probability that performing arithmetic over the prime field $\mathbb{F}_p$ when p is chosen randomly from $[1, p_m]$, introduces any spurious linear equalities is bounded above by $\frac{n}{2^a}$, if $p_m = 2^{a+1} b_m' \log(2^a b_m')$, where $b_m' = (b_m + 1)(k_v + 1)^2$.*

The proof of Theorem 6 can be found in [7]. We sketch here the main idea. It can be shown that an appropriate deterministic summary based algorithm (along the lines of one described in [14]) can represent the numerator and denominator of the value of variables in different states using at most $b_m = 1 + n_{max}(n_{max}+1)(\log s + \log c_m)$ bits. We then observe that no spurious linear equalities are introduced at a given program point if the determinant of an appropriate matrix, with at most $k_v + 1$ rows and columns and with entries that require at most b_m bits for representation, remains non-zero when the arithmetic operations are performed over $\mathbb{F}_p$. The probability of such an event happening is given by the following property.

PROPERTY 7. *The probability that two distinct b bit integers are equal modulo a randomly chosen prime from $[1, p_m]$ is at most $\frac{1}{u}$ for $p_m \ge 2ub \log(ub)$.*

The proof of Property 7 follows from the prime number theorem which states that the number of prime numbers less than x is at least $\frac{x}{\log x}$.

THEOREM 8. *Suppose the prime p' used for verifying the linear equalities discovered in the first round is chosen randomly from $[1, p_m]$. Let c_m' be the bound on the absolute value of the coefficients of the linear equalities discovered in the first round. Then, the probability of incorrect verification of the coefficients is bounded above by $\frac{nk_v}{2^a}$, if $p_m = 2^{a+1} b_m' \log(2^a b_m')$, where $b_m' = (b_m + c_m')(k_v + 1) + \log(k_v + 1)$.*

The proof of Theorem 8 is also based on Property 7 and can be found in [7].

Theorem 6 and Theorem 8 suggest that the random interpreter must perform arithmetic with primes that require $O(nk_v k_i \log n)$ bits for representation (assuming s and c_m to be constants). However, we feel that this is a conservative analysis, and we do not know of any program that illustrates this worst-case behavior. Experiments discussed in Section 10 suggest that even 32-bit primes do not yield any error in practice.

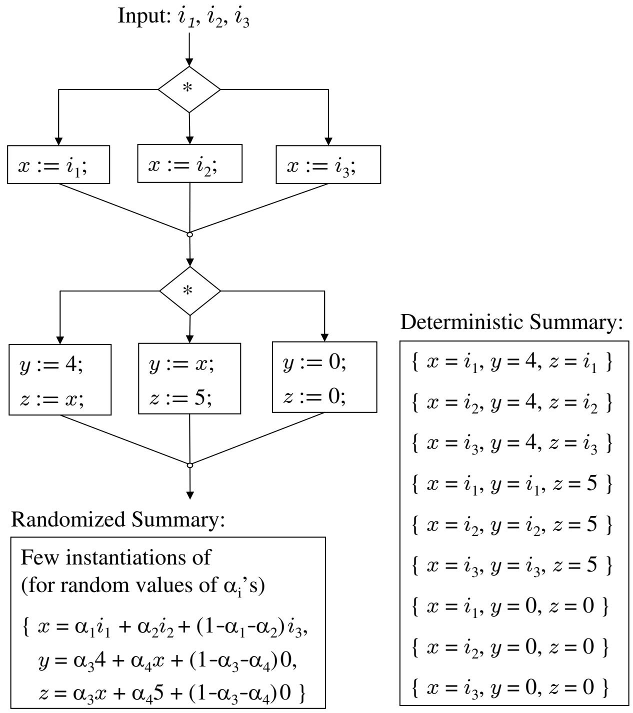


Figure 8: Illustration of the difference between the deterministic summary computed by MOS algorithm and the randomized summary computed by our algorithm.

8.3 Computational Complexity

As discussed in Section 7, the inter-procedural random interpreter has a complexity of $O(n_{\mathtt{s}} k_{\mathtt{v}} k_{\mathtt{i}}^2 t + n_{\mathtt{p}} k_{\mathtt{v}} k_{\mathtt{i}}^2 k_{\mathtt{o}} t^2)$ (assuming unit cost for each arithmetic operation). It follows from Theorem 3 that for probabilistic soundness, we need to choose t to be greater than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$. However, we feel that our analysis for probabilistic soundness is conservative. Experiments discussed in Section 10 suggest that even $t = 3$ does not yield any error in practice if we want to verify equalities (among any number of program variables), or discover equalities between 2 program variables.

8.4 Related Work

Recently, Muller-Olm and Seidl gave a deterministic algorithm (MOS) that discovers all linear equalities in programs that have been abstracted using non-deterministic conditionals [14]. The MOS algorithm is also based on computing summaries of procedures. However, their summaries are deterministic and consist of linearly independent runs of the program. The program shown in Figure 8 illustrates the difference between the deterministic summaries computed by MOS algorithm and the randomized summaries computed by our algorithm. The MOS algorithm maintains the (linearly independent) real runs of the program, and it may have to maintain as many as $k_{\mathtt{v}}(k_{\mathtt{i}}+1)$ runs. The runs maintained by our algorithm are fictitious as they do not arise in any concrete execution of the program; however they have the property that (with high probability over the random choices made by the algorithm) they entail exactly the same set of equivalences in all contexts as do the real runs. Our algorithm needs to maintain only a few runs. The conservative theoretical bounds show that more than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$ runs are required, while experiments suggest that even 3 runs are good enough (if we want to verify linear equalities, or discover linear equalities between 2 program variables).

The authors have proved a complexity of $O(nk_{\mathtt{v}}^8)$ for the MOS algorithm assuming a unit cost measure for arithmetic operations. It turns out that the arithmetic constants that arise in MOS algorithm may be large enough that $\Omega(2^n)$ bits for required for representing constants, and hence $\Omega(2^n)$ time is required for performing a single arithmetic operation. The program shown in Figure 6 illustrates such an exponential behavior of MOS algorithm. The MOS algorithm can also use the technique of avoiding large arithmetic constants by performing arithmetic modulo a randomly chosen prime. However this makes MOS a randomized algorithm; and the complexity of our randomized algorithm remains better than that of MOS. It is not clear if there exists a polynomial time deterministic algorithm for this problem.

Sagiv, Reps and Horwitz gave an efficient algorithm (SRH) to discover linear constants inter-procedurally in a program [20]. Their analysis considers only those affine assignments whose right hand sides contain at most one occurrence of a variable. However, our analysis is more precise as it treats all affine assignments in a precise manner, and also it discovers all linear equalities (not just constants).

The first intra-procedural analysis for discovering linear equalities was given by Karr way back in 1976 [11]. The fact that it took several years to obtain an inter-procedural analysis to discover all linear relationships in programs that have been abstracted using linear arithmetic assignments demonstrates the complexity of inter-procedural analysis.

9. SPECIAL CASE OF UNINTERPRETED FUNCTIONS

In this section, we discuss the use of inter-procedural random interpretation for discovering Herbrand equivalences among program sub-expressions that have been abstracted using unary uninterpreted functions. This abstraction is useful for modeling fields of data-structures and can be used to compute must-alias information.

Note that we restrict our attention to unary uninterpreted functions (instead of considering the more general binary uninterpreted functions). This is because in the case of binary uninterpreted functions, expressions are mapped to vectors rather than scalars. The size of these vectors is linearly proportional to the depth of any expression computed by the program along any acyclic path [9]. In an inter-procedural setting, the depth of such expressions can be exponential in the size of the program. Hence, unless we can prove the conjecture that the size of the vectors need only be logarithmic in the size of the program (as mentioned in [9]), the complexity of processing each node will be exponential in the size of the program, which is perhaps not any worst-case better than a non-summary based inter-procedural analysis (which involves reducing the problem of inter-procedural analysis to intra-procedural analysis by doing procedure inlining as described in Appendix B). Since we are interested in polynomial-time complexity algorithms in this paper, we leave out the discussion of the complexity of the inter-procedural analysis for binary uninterpreted functions. However, the inter-procedural random interpretation technique described in this paper is applicable to reasoning about binary uninterpreted functions too.

9.1 Error Probability Analysis

Since the `SEval` function for unary uninterpreted functions contains random variables, the general case in Theorem 3 applies, which specifies a bound on the error probability for verification of one equivalence. The total error probability of the random interpreter is given by the product of this error probability (for verification of one equivalence) with the number of equivalences between program sub-expressions verified by the random interpreter.

For probabilistic soundness, Theorem 3 requires choosing t to be greater than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$. However, for the specific case of unary uninterpreted functions, we require t to be only greater than 6. This is because of the following reason. Observe that any equivalence in the abstraction of unary uninterpreted functions involves only 2 program variables. Also, observe that the validity of any equivalence (at any program point) depends on the relationship between at most 2 input variables of the enclosing procedure. Hence, the proof of Theorem 3 can be specialized to the specific case of unary uninterpreted functions by substituting $k_{\mathtt{v}} = 2$ and $k_{\mathtt{i}} = 2$, which yields the desired constraint that t need only be greater than 6.

9.2 Computational Complexity

It follows from the above discussion that for probabilistic soundness, we need to choose t to be greater than 6. This yields a total complexity of $O(n_{\mathtt{s}} k_{\mathtt{v}} k_{\mathtt{i}}^2 + n_{\mathtt{p}} k_{\mathtt{v}} k_{\mathtt{i}}^2 k_{\mathtt{o}})$ for the random interpreter (assuming unit cost for each arithmetic operation).

9.3 Related Work

There have been number of attempts at developing intra-procedural algorithms for global value numbering, which is the problem of discovering equivalences in a program with non-deterministic conditionals and uninterpreted functions. Until recently, all the known algorithms were either exponential or incomplete [12, 18, 19]. Recently, we presented a randomized [9] as well as a deterministic [10] polynomial time complete algorithm for this problem.

Recently, Müller-Olm, Seidl, and Steffen have given an algorithm to detect Herbrand equalities in an inter-procedural setting [15]. Their algorithm is complete (i.e., it detects all valid Herbrand equalities) for side-effect-free procedures that have only one return value. Their algorithm can also detect all Herbrand constants. In contrast, our random interpretation based inter-procedural analysis detects all equivalences without any restriction on the number of return values, or global values affected by a procedure. However, our algorithm has a polynomial time complexity bound only for unary uninterpreted functions.

Gil and Itai have characterized the complexity of a similar problem, that of type analysis of object oriented programs in an inter-procedural setting [6].

10. EXPERIMENTS

In this paper, we have expanded the body of theoretical evidence that randomized algorithms have certain advantages, such as simpler implementations and better computational complexity, over deterministic ones. We now describe our experience with experimenting some of these algorithms.

t	$p = 983$			$p = 65003$			$p = 268435399$		
	E_1	E_2	E_3	E_1	E_2	E_3	E_1	E_2	E_3
2	1.7	0.2	95.5	0.1	0	95.5	0	0	95.5
3	0	0	64.3	0	0	3.2	0	0	0
4	0	0	0.2	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0

E_1 : % of incorrect variable constants $x = c$ reported
E_2 : % of incorrect variable equalities $x = y$ reported
E_3 : % of incorrect dependent induction variables reported

Table 1: Percentage E_i of incorrect relationships of different kinds discovered by the inter-procedural random interpreter as a function of the number of runs t and the randomly chosen prime p on a collection of programs, which are listed in Table 2. For example, with $t = 2$ and $p = 983$, the random interpreter discovered 3501 variable constants, of which 59 were incorrect; hence, $E_1 = \frac{59}{3501-59} \times 100 \approx 1.7\%$. The total number of correct relationships discovered were 3442 variable constants, 4302 variable equalities, and 50 dependent induction variables.

The goals of these experiments are threefold: (1) measure experimentally the soundness probability and its variation with certain parameters of the algorithm, (2) measure the running time and effectiveness of the inter-procedural version of the algorithm, and compare it to the intra-procedural version, and (3) perform a similar comparison with a deterministic inter-procedural algorithm.

We ran all experiments on a Pentium 1.7GHz machine with 1Gb of memory. We used a number of programs, up to 28,000 lines long, some from the SPEC95 benchmark suite, and others from similar measurements in previous work [20]. We measured running time using enough repetitions to avoid timer resolution errors.

We have implemented the inter-procedural algorithm described in this paper, in the context of the linear equalities domain. The probability of error grows with the complexity of the relationships we try to find, and shrinks with the increase in number of runs and the size of the prime number used for modular arithmetic. The last two parameters have a direct impact on the running time.[5]

We first ran the inter-procedural randomized analysis on our suite of programs, using a variable number of runs, and prime numbers of various sizes. We consider here equalities with constants (x=c), variable equalities (x=y), and linear induction variable dependencies among variables used and modified in a loop (dep).[6] Table 1 shows the number of erroneous relationships reported in each case, as a percentage of the total relationships found for the corresponding kind.

These results are for programs with hundreds of variables; and our analysis for probabilistic soundness requires $t > k_{\mathtt{v}} + 2k_{\mathtt{i}}$, yet in practice we do not notice any errors for $t \geq 4$. Similarly, our theoretical estimates of the error prob-

[5]For larger primes, the arithmetic operations cannot be inlined anymore.
[6]We found many more linear dependencies, but report only the induction variable ones because those have a clear use in compiler optimization.

Program	Size	Random Inter-procedural						Random Intra-procedural				Det. Inter-procedural	
		inp	x=c	x=y	dep	Time_2	Time_3	Δx=y	Δx=c	Δdep	Speedup_3	Δinp	Speedup_2
go	29K	63	1700	796	6	47.3	70.4	170	260	3	107	17	1.9
ijpeg	28K	31	825	851	12	3.8	5.7	34	1	9	24	3	2.3
li	23K	53	392	2283	9	34.0	51.4	160	1764	6	756	20	1.3
gzip	8K	49	525	372	2	2.0	3.05	200	12	1	39	6	2.0
bj	2K	0	117	9	0	1.2	1.8	0	0	0	11	0	2.3
linpackc	2K	14	86	16	1	0.07	0.11	17	1	1	9	0	1.8
sim	2K	3	117	296	0	0.35	0.54	3	11	0	22	0	1.7
whets	1K	9	80	2	6	0.03	0.05	17	1	0	9	0	1.5
flops	1K	0	52	4	4	0.02	0.03	0	0	0	22	0	2.0

Size:	# of lines of C-code
inp:	# of linear relationships among input variables at start of procedures
x=c:	# of variables equal to constant values
x=y:	# of variable equalities
dep:	# of dependent loop induction variables
Time_i:	Time (in seconds) for $r = i$ runs
Δ k:	Difference of # of relationships of kind k found by Random Inter-procedural and given algorithm
Speedup_i:	Ratio of time with Time_i of Random Inter-procedural

Table 2: Comparison of precision and efficiency between the randomized inter-procedural, randomized intra-procedural, and deterministic inter-procedural analyses on SPEC benchmarks.

ability when using small primes are also pessimistic. With the largest prime shown in Table 1, we did not find any errors if we use at least 3 runs.[7] In fact, for the problem of discovering simpler kinds of equalities (variable constants $x = c$, variable equalities $x = y$), we do not observe any errors for $t = 2$. This is in fact the setup that we used for the experiments described below that compare the precision and cost (in terms of time) of the randomized inter-procedural analysis with that of randomized intra-procedural analysis and deterministic inter-procedural analysis.

The first set of columns in Table 2 show the results of the inter-procedural randomized analysis for a few benchmarks with more than 1000 lines of code each. The column headings are explained in the caption. We ran the algorithm with both $t = 2$ and $t = 3$, since the smaller value is faster and sufficient for discovering equalities between variables and constants. As expected, the running time increases linearly with t. The noteworthy point here is the number of relationships found between the input variables of a procedure.

In the second set of columns in Table 2 we show how many fewer relationships of each kind are found by the intra-procedural randomized analysis, and how much faster that analysis is, when compared to the inter-procedural one. The intra-procedural analysis obviously misses all of the input relationships and consequently misses some internal relationships as well, but it is much faster. The loss of effectiveness results (when performing an inter-procedural analysis as compared to an intra-procedural analysis) are similar to those reported in [20]. Whether the additional information generated by the inter-procedural analysis is worth the extra implementation and compile-time cost will depend on how that information is to be used. For compiler optimization it is likely that intra-procedural results are good enough, but perhaps for applications such as program verification the extra cost might be worth paying.

Finally, we compare our inter-procedural random interpretation based algorithm with an inter-procedural deterministic algorithm. We have implemented and experimented with the SRH algorithm [20], and the results are shown in the third set of columns in Table 2. SRH is less precise than our algorithm, in that it searches only for equalities with constants ($x = c$). It does indeed find all such equalities that we also find. In theory, there are equalities with constants that we can find but SRH cannot, because they are consequences of more complex linear relationships. However, the set of benchmarks that we have looked at does not seem to have any such hard-to-find equalities. For comparison with this algorithm, we used $t = 2$, which is sufficient for finding equalities of the form $x = c$ and $x = y$. However, we find a few more equalities between the input variables (Δ inp), and numerous equalities between local variables, which SRH does not attempt to find. On average, SRH is 1.5 to 2.3 times faster than our algorithm. Again, the cost may be justified by the expanded set of relationships that we discover.

A fairer comparison would have been with the MOS algorithm [14], which is as precise as our inter-procedural randomized algorithm. However, implementing this algorithm seems quite a bit more complicated than either of our algorithm or SRH. We also could not obtain an implementation from anywhere else. Furthermore, we speculate that due to the fact that MOS requires data structures whose size is $O(k_v^4)$ at every program point, it will not fare well on the larger examples that we have tried, which have hundreds of variables and tens of thousands of program points. Another source of bottleneck may be the complexity of manipulating large constants that may arise during its analysis.

11. CONCLUSION

We described a unified framework for random interpretation, along with generic completeness and probabilistic soundness theorems, both for verifying and for discovering relationships among variables in a program. These results can be instantiated directly to the domain of linear relation-

[7] With only 2 runs, we find a linear relationship between any pair of variables, as expected.

ships and, separately, of Herbrand equivalences, to derive existing algorithms and their properties. This framework also provides guidance for instantiating the algorithms to other domains. It is, however, an open problem if a complete algorithm can be obtained for a combination of domains, such as linear arithmetic and Herbrand equivalences.

The most important result of this paper is to show that random interpreters can be extended in a fairly natural way to an inter-procedural analysis. This extension is based on the observation that a summary of a procedure can be stored concisely as the results of a number of intra-procedural random interpretations with symbolic values for input variables. Using this observation, we have obtained inter-procedural randomized algorithms for linear relationships (with better complexity than similar deterministic algorithms) and for Herbrand equivalences (for which there is no deterministic algorithm).

Previously published random interpretation algorithms resemble random testing procedures, from which they inherit trivial data structures and low complexity. The algorithms described in this paper start to mix randomization with symbolic analysis. The data structures become somewhat more involved, essentially consisting of random instances of otherwise symbolic data structures. Even the implementation of the algorithms starts to resemble that of symbolic deterministic algorithms. This change of style reflects our belief that the true future of randomization in program analysis is not in the form of a world parallel to traditional symbolic analysis algorithms, but in an organic mixture that exploits the benefits of both worlds.

12. REFERENCES

[1] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, June 1997.

[2] M. C. Cary. Lattice basis reduction algorithms and applications. Unpublished survey paper. Feb. 2002.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1990.

[5] C. F. Gauss. *Disquisitiones Arithmeticae*. Article 171, English Edition (Translated by A. Clarke). Springer-Verlag, New York, 1986.

[6] J. Y. Gil and A. Itai. The complexity of type analysis of object oriented programs. *Lecture Notes in Computer Science*, 1445:601–634, 1998.

[7] S. Gulwani. Program analysis using random interpretation. Ph.d. dissertation, Computer Science Department, University of California at Berkeley, 2005.

[8] S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In *30th ACM Symposium on Principles of Programming Languages*, pages 74–84. ACM, Jan. 2003.

[9] S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *31st ACM Symposium on Principles of Programming Languages*, pages 342–352, Jan. 2004.

[10] S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *11th Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*. Springer, 2004.

[11] M. Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151. Springer, 1976.

[12] G. A. Kildall. A unified approach to global program optimization. In *1st ACM Symposium on Principles of Programming Language*, pages 194–206. ACM, Oct. 1973.

[13] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec. 1992.

[14] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *31st Annual ACM Symposium on Principles of Programming Languages*, pages 330–341. ACM, Jan. 2004.

[15] M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural herbrand equalities. In *Proceedings of the European Symposium on Programming*, LNCS. Springer-Verlag, 2005.

[16] T. Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica*, 33(8):739–757, Nov. 1996.

[17] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM Symposium on POPL*, pages 49–61. ACM, 1995.

[18] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *15th ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[19] O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 1999.

[20] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 30 Oct. 1996.

[21] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *JACM*, 27(4):701–717, Oct. 1980.

[22] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors,. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[23] R. Zippel. Probabilistic algorithms for sparse polynominals. In *Proceedings of the International Symposium on Symbolic and Algebraic Manipulation (EUROSAM)*, volume 72 of *LNCS*, pages 216–226. Springer, June 1979.

APPENDIX

A. DEFINITION AND PROPERTIES OF FULLY SYMBOLIC STATE

Phase 1

We hypothetically extend the random interpreter to also compute a symbolic state $\tilde{\rho}$, paths A, and an integer d (also referred to as d_S) at each program point besides a sample S. Paths A represent the set of all paths analyzed by the random interpreter inside the corresponding procedure. A path is simply a sequence of assignments. The symbolic state $\tilde{\rho}$ gives for each variable x that polynomial whose different random instantiations are the values of x in different states in the sample S. The integer d represents the maximum degree of the weight variables (introduced at join nodes and procedure call nodes) in any polynomial in the symbolic state $\tilde{\rho}$. d also represents the maximum number of join points and procedure calls analyzed by the random interpreter along any path immediately after computation of sample S. The values of $\tilde{\rho}$, A and d are updated for each flowchart node as shown below.

Initialization. Following initialization is done at procedure entry points.

$$\begin{aligned} \tilde{\rho}(x) &= \texttt{SEval}(x) \\ A &= \{\epsilon\} \\ d &= 0 \end{aligned}$$

ϵ denotes the empty sequence of assignment nodes. At all other program points, the following initialization is done: $\tilde{\rho} = \perp$, $A = \emptyset$, and $d = -1$.

Assignment Node. See Figure 1 (a).
If $\tilde{\rho}' = \perp$, then $\tilde{\rho} = \perp$, $A = \emptyset$ and $d = -1$. Else,

$$\begin{aligned} \tilde{\rho} &= \tilde{\rho}'[x \leftarrow (\texttt{SEval}(e))[\tilde{\rho}']] \\ A &= \{a; x := e \mid a \in A'\} \\ d &= d' \end{aligned}$$

$(\texttt{SEval}(e))[\tilde{\rho}']$ refers to the polynomial obtained from $\texttt{SEval}(e)$ by replacing all variables y by $\tilde{\rho}'(y)$.

Non-deterministic Assignment Node. See Figure 1 (b).
If $\tilde{\rho}' = \perp$, then $\tilde{\rho} = \perp$, $A = \emptyset$ and $d = -1$. Else,

$$\begin{aligned} \tilde{\rho} &= \tilde{\rho}'[x \leftarrow x'], \text{ where } x' \text{ is a fresh variable} \\ A &= \{a; x :=? \mid a \in A'\} \\ d &= d' \end{aligned}$$

Non-deterministic Conditional Node. See Figure 1 (c).

$$\begin{aligned} \tilde{\rho}^1 = \tilde{\rho}^2 &= \tilde{\rho} \\ A^1 = A^2 &= A \\ d^1 = d^2 &= d \end{aligned}$$

Join Node. See Figure 1 (d).
If $\tilde{\rho}^1 = \perp$, then $\tilde{\rho} = \tilde{\rho}^2$, $A = A^2$ and $d = d^2$. Else if $\tilde{\rho}^2 = \perp$, then $\tilde{\rho} = \tilde{\rho}^1$, $A = A^1$ and $d = d^1$. Else,

$$\begin{aligned} \tilde{\rho}(x) &= \alpha\tilde{\rho}^1(x) + (1-\alpha)\tilde{\rho}^2(x), \text{ for all variables } x \\ A &= A^1 \cup A^2 \\ d &= max(d^1, d^2) + 1 \end{aligned}$$

α is a fresh variable that does not occur in $\tilde{\rho}^1$ and $\tilde{\rho}^2$.

Procedure Call. See Figure 1 (e).
If $\tilde{\rho}' = \perp$ or $Y_{P'} = \perp$, then $\tilde{\rho} = \perp$, $A = \emptyset$ and $d = -1$. Else,

$$\begin{aligned} \tilde{\rho}(x) &= \begin{cases} \tilde{Y}_{P'}(x)[\tilde{\rho}'(y_1)/y_1, .., \tilde{\rho}'(y_k)/y_k] & \text{if } x \in O_{P'} \\ \tilde{\rho}'(x) & \text{otherwise} \end{cases} \\ A &= \{a_1; a_2 \mid a_1 \in A', a_2 \in \mathit{Paths}(Y_{P'})\} \\ d &= d' + 1 \end{aligned}$$

where,

$$\tilde{Y}_{P'} = \sum_{j=1}^{t-1} \alpha_j Y_{P',j} + (1 - \sum_{j=1}^{t-1} \alpha_j) Y_{P',t}$$

α_j's are fresh variables that do not occur in $\tilde{\rho}'$ and $Y_{P',j}$. $O_{P'}$ is the set of output variables of procedure P' and $y_1, \ldots, y_k$ are the input variables of procedure P'. $\mathit{Paths}(Y_P)$ refers to be the set of paths A after the exit node of procedure P during computation of the set of runs Y_P for procedure P. Initially, Y_P is defined to be $\perp$ and $\mathit{Paths}(Y_P)$ is defined to be the empty set for all procedures P.

We say that a summary Y_P is sound and complete for a context C when $Y_P \models_C e_1 = e_2 \iff \mathit{Holds}(e_1 = e_2, \mathit{Paths}(Y_P), C)$.

LEMMA 9. *Suppose that the summaries of all procedures plugged into analyzing a procedure P are sound and complete for all contexts. Let $\tilde{\rho}$ be the fully-symbolic state and A be the set of paths computed by the random interpreter at any program point inside procedure P (in phase 1). Let C be any context for procedure P and $e_1 = e_2$ be some equivalence. Then,*

$$\tilde{\rho} \models_C e_1 = e_2 \iff \mathit{Holds}(e_1 = e_2, A, C)$$

PROOF. The proof of the lemma is by induction on the number of flowchart nodes analyzed by the random interpreter. The base case follows easily from the soundness and completeness properties (properties B1 and B2) of the SEval function. For the inductive case, the proof is trivial if one of the inputs of a node is $\perp$; hence we consider the scenarios when all inputs to a node are non-$\perp$.

Assignment Node. See Figure 1 (a).
Let $e_1' = e_1[^e/_x]$ and $e_2' = e_2[^e/_x]$. Note that $e_1' = e_2'$ is the weakest precondition of $e_1 = e_2$ immediately before the assignment node along paths in A. Hence,

$$\mathit{Holds}(e_1 = e_2, A, C) \iff \mathit{Holds}(e_1' = e_2', A', C)$$

It follows from the induction hypothesis on $\tilde{\rho}'$ that

$$\mathit{Holds}(e_1' = e_2', A', C) \iff \tilde{\rho}' \models_C e_1' = e_2'$$

It now follows from property B3 of the SEval function that

$$\tilde{\rho}' \models_C e_1' = e_2' \iff \tilde{\rho} \models_C e_1 = e_2$$

Non-deterministic Assignment Node. See Figure 1 (b).
Let x' be the fresh variable assigned to x by the symbolic random interpreter in obtaining state $\tilde{\rho}$ from $\tilde{\rho}'$. Let $e_1' = e_1[^{x'}/_x]$ and $e_2' = e_2[^{x'}/_x]$. Note that $e_1' = e_2'$ is the weakest precondition of $e_1 = e_2$ along paths in A immediately before the assignment node. The rest of the proof is now similar to the case of assignment node above.

Non-deterministic Conditional Node. See Figure 1(c). This case is trivial since $\tilde{\rho}^1 = \tilde{\rho}$ and $\tilde{\rho}^2 = \tilde{\rho}$.

Join Node. See Figure 1(d).
Note that

$$\begin{aligned} Holds(e_1 = e_2, A, C) \iff & Holds(e_1 = e_2, A^1, C) \\ & \text{and } Holds(e_1 = e_2, A^2, C) \end{aligned}$$

It follows from the induction hypothesis on $\tilde{\rho}^1$ and on $\tilde{\rho}^2$ that

$$\begin{aligned} Holds(e_1 = e_2, A^1, C) & \iff & \tilde{\rho}^1 \models_C e_1 = e_2 \\ Holds(e_1 = e_2, A^2, C) & \iff & \tilde{\rho}^2 \models_C e_1 = e_2 \end{aligned}$$

Since $\tilde{\rho}' = \alpha\tilde{\rho}^1 + (1-\alpha)\tilde{\rho}^2$, the following holds:

$$\tilde{\rho}' \models_C e_1 = e_2 \iff \tilde{\rho}^1 \models_C e_1 = e_2 \text{ and } \tilde{\rho}^2 \models_C e_1 = e_2$$

Procedure Call. See Figure 1 (e).
Let $\tilde{\rho}_j$ be the following symbolic state:

$$\tilde{\rho}_j(x) \quad = \quad \begin{cases} Y_{P',j}(x)[\tilde{\rho}'(y_1)/y_1, .., \tilde{\rho}'(y_k)/y_k] & \text{if } x \in O_{P'} \\ \tilde{\rho}'(x) & \text{otherwise} \end{cases}$$

where $O_{P'}$ is the set of output variables of procedure P' and $y_1, \ldots, y_k$ are the input variables of procedure P'. It follows from the induction hypothesis on $\tilde{\rho}'$ and the soundness and completeness of the summary for procedure P' that

$$Holds(e_1 = e_2, A, C) \quad \iff \quad \forall j \in \{1, .., t\},\ \tilde{\rho}_j \models_C e_1 = e_2$$

Since $\tilde{\rho}' = \sum_{j=1}^{t-1} \alpha_j \tilde{\rho}_j + (1 - \sum_{j=1}^{t-1} \alpha_j)\tilde{\rho}_t$, the following holds:

$$\tilde{\rho} \models_C e_1 = e_2 \iff \forall j \in \{1, .., t\},\ \tilde{\rho}_j \models_C e_1 = e_2$$

□

Phase 2

We hypothetically extend the random interpreter to also compute a symbolic state $\tilde{\rho}$, paths A, and an integer d (also referred to as d_S) at each program point besides a sample S, as is done in proving the correctness of phase 1. These are updated for different flowchart nodes as in phase 1, except for the initialization of any procedure other than `Main`.

The entry point of any procedure P other than `Main` is initialized as follows. Let there be m call sites for procedure P that have a non-$\perp$ sample. Let $A^i, d^i, \tilde{\rho}^i$ be the values computed by the random interpreter at the i^{th} such call site. Then, the random interpreter performs the following initialization for the entry point of procedure P.

$$\begin{aligned} A &= A^1 \cup .. \cup A^m \\ \tilde{\rho} &= \sum_{i=1}^{m-1} \alpha_i \tilde{\rho}^i + (1 - \sum_{i=1}^{m-1} \alpha_i)\tilde{\rho}^m \\ d &= max(d^1, .., d^m) + 1 \end{aligned}$$

Here $\alpha_1, \ldots, \alpha_{m-1}$ are fresh variables.

We use the notation $Holds_2(e_1 = e_2, A)$ to denote that the equivalence $e_1 = e_2$ holds at the end of all paths in A. We also use the notation $\rho \models e_1 = e_2$ to denote that `Eval`$(e_1, \rho) =$ `Eval`(e_2, ρ) for any state ρ.

LEMMA 10. *Suppose that the summaries of all procedures plugged into analyzing a procedure P are sound and complete for all contexts. Let $\tilde{\rho}$ be the fully-symbolic state and A be the set of paths computed by the random interpreter at any program point inside procedure P (in phase 2). Let $e_1 = e_2$ be any equivalence. Then,*

$$\tilde{\rho} \models e_1 = e_2 \quad \iff \quad Holds_2(e_1 = e_2, A)$$

PROOF. The proof is by induction on the number of flowchart nodes analyzed by the random interpreter. The base case follows easily from the soundness and completeness of the `SEval` function. For the inductive case, the proofs for assignment node (both deterministic and non-deterministic), non-deterministic conditional node, join node, and procedure call are similar to the ones for phase 1. We now prove the inductive case for the entry point of a procedure P. Let $\pi^1, .., \pi^m$ be the program points immediately before the calls to procedure P that have a non-$\perp$ sample. Let $\tilde{\rho}^i, A^i, d^i$ be the values computed by the random interpreter at those points. The following holds:

$$Holds_2(e_1 = e_2, A) \iff \forall i \in \{1, .., m\}, Holds_2(e_1 = e_2, A^i)$$

It follows from the induction hypothesis on $\tilde{\rho}^i$ that

$$Holds_2(e_1 = e_2, A^i) \quad \iff \quad \tilde{\rho}^i \models e_1 = e_2$$

Since $\tilde{\rho} = \sum_{i=1}^{m-1} \alpha_i \tilde{\rho}^i + (1 - \sum_{j=1}^{m-1})\alpha_m \tilde{\rho}^m$, the following holds:

$$\tilde{\rho} \models e_1 = e_2 \iff \forall i \in \{1, .., m\}\ \tilde{\rho}^i \models e_1 = e_2$$

□

B. EQUIVALENT PROGRAM WITHOUT ANY PROCEDURE CALLS

In this section, we show how to convert each procedure in a program (in our program model) into an equivalent procedure (in the sense that both procedures satisfy the same set of equivalences at corresponding program points) that does not use any procedure calls. It follows from Theorem 4 and Theorem 5 that any program node is processed at most H_1 times in phase 1 and H_2 times in phase 2 during fixed-point computation. We use this observation to transform the given program (with procedure calls) in the following manner:

1. In the original program, unroll all loops inside procedures and in the call graph along the paths taken by any standard summary based inter-procedural analyzer in phase 1. It follows from Theorem 4 that this unrolling leads to an H_1 times increase in the size of the procedures.
2. In the acyclic call graph with acyclic procedures obtained in step 1, replace all procedure calls by recursive procedure inlining in a bottom-up manner.

The procedures thus obtained have at most m^m nodes, where m is the size of the program obtained in step 1 (measured in terms of the number of nodes). Thus, the size of each procedure is bounded above by $(nH_1)^{nH_1}$ nodes.

Next, observe that there exists a generalized context (which is an acyclic program fragment) for each procedure P in the sense that whatever equivalences hold among the input variables of P in all calls to P, the same set of equivalences hold among those variables at the end of the generalized context. The generalized context can be obtained as follows:

3. In the original program, unroll all loops in the call graph and inside procedures along the paths taken by any standard summary based inter-procedural analyzer in phase 2. It follows from Theorem 5 that this unrolling increases the size of the procedures by a factor of at most H_2.

4. In the acyclic call graph with acyclic procedures thus obtained in step 3, build a context (in a bottom-up manner) for a procedure P, which has q call sites inside procedures $P_1, \ldots, P_q$, as follows. Construct a non-deterministic conditional with q branches, whose i^{th} branch is the context of procedure P_i followed by the code of P_i that leads up to the corresponding call to procedure P.

The contexts thus obtained have a worst-case size of m^m, where m is the size of the program obtained in step 3. This leads to a total size of $(nH_2)^{nH_2}$ for each context.

We can now obtain the desired program without any procedure calls as follows:

5. In the original program, prepend all procedures by their generalized contexts obtained in step 4. This leads to a total size of at most $2(nH_2)^{nH_2}$ nodes for each procedure.

6. In the program obtained in step 5, replace all procedure calls by their summaries obtained in step 2. This leads to a total size of at most $n_{max} = 2(nH_1)^{nH_1}(nH_2)^{nH_2}$ nodes for each procedure.

C. PROOF OF LEMMA 1

Let S be any sample and $\tilde{\rho}$ be the corresponding fully-symbolic state of k variables computed by the random interpreter at some program point.

For any state ρ of k variables, let $J(\rho)$ denote the vector $(\rho(x_1), \ldots, \rho(x_k), 1)$. Let A be the set of elements of the vectors in the set $\{J(\tilde{\rho}')[v_i/w_i] \mid v_i \in \mathbb{F}_p\}$, where $J(\tilde{\rho}')[v_i/w_i]$ denotes the vector obtained from $J(\tilde{\rho}')$ by replacing all weight variables w_i by some choices of elements v_i from $\mathbb{F}_p$. Let $\mathbb{F}'_p$ be the smallest field generated by the elements in set A. Let $\mathcal{U}(\tilde{\rho}')$ be the vector space generated by the vectors $\{J(\tilde{\rho}')[v_i/w_i] \mid v_i \in \mathbb{F}_p\}$ over the field $\mathbb{F}'_p$. Let m be the rank of this vector space. Note that $m \leq 1+k$ (since there can be at most $1+k$ linearly independent vectors over $\mathbb{F}'_p$, where each vector consists of $1+k$ elements from $\mathbb{F}'_p$).

Let $\mathcal{E}$ be the event that the vectors $J(S'_1), \ldots, J(S'_t)$ have less than m linearly independent vectors over the field $\mathbb{F}'_p$. We partition the event $\mathcal{E}$ into disjoint cases depending on which of the vectors $J(S'_1), \ldots, J(S'_t)$ are linearly independent. Let I be any subset of $\{1, .., t\}$. Let $\mathcal{F}_i$ be the event that $J(S'_i)$ is linearly independent of $J(S'_1), \ldots, J(S'_{i-1})$. Let $\mathcal{E}_I$ be the event $\bigwedge_{i \in I} \mathcal{F}_i \wedge \bigwedge_{i \in \{1,\ldots,t\}-I} \neg\mathcal{F}_i$. The set of events $\{\mathcal{E}_I \mid I \subseteq \{1, .., t\}, 1 \in I, |I| < m\}$ is a disjoint partition of the probability space for event $\mathcal{E}$. Thus,

$$\Pr(\mathcal{E}) = \sum_{I \subseteq \{1,..,t\}, 1 \in I, |I| < m} \Pr(\mathcal{E}_I)$$

$$\begin{aligned}
&\Pr(\mathcal{E}_I) \\
&= \Pr\Big(\bigwedge_{i \in I} \mathcal{F}_i \;\wedge \bigwedge_{i \in \{1,..,t\}-I} \neg\mathcal{F}_i\Big) \\
&= \prod_{i \in I} \Pr\Big(\mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \;\wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j\Big] \\
&\quad \times \prod_{i \in \{1,..,t\}-I} \Pr\Big(\neg\mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \;\wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j\Big) \\
&\leq \prod_{i \in \{1,..,t\}-I} \Pr\Big(\neg\mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \;\wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j\Big)
\end{aligned}$$

We now bound $\Pr(\neg\mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j)$ for any $i \in \{1, .., t\} - I$. Let I_i be the set $\{j \in I \mid j < i\}$ and let $n_i = |I_i|$. Let M_i be the matrix with $n_i + 1$ rows from the set $\{J(S'_j) \mid j \in I_i \cup \{i\}\}$. Let $\tilde{M}_i$ be the matrix with $n_i + 1$ rows obtained from M_i by replacing the row $J(S'_i)$ with $J(\tilde{\rho}')$. Since the events $\{\mathcal{F}_j\}_{j \in I_i}$ occur, the vectors $\{J(S'_j)\}_{j \in I_i}$ are linearly independent. Note that $J(\tilde{\rho}')$ is linearly independent of the vectors $\{J(S'_j)\}_{j \in I_i}$ (because otherwise $Rank(\mathcal{U}(\tilde{\rho}'))$ would be n_i, which is less than m). Hence, $Rank(\tilde{M}_i) = n_i + 1$. Thus, there exists a submatrix $\tilde{M}^s_i$ of $\tilde{M}_i$ of size $(n_i+1) \times (n_i+1)$ such that $Rank(\tilde{M}^s_i) = n_i + 1$, or equivalently, $Det(\tilde{M}^s_1) \not\equiv 0$. Let M^s_i be the submatrix of M_i consisting of those columns of M_i that are used in obtaining $\tilde{M}^s_i$ from $\tilde{M}_i$. Note that S'_i is a random instantiation of $\tilde{\rho}'$, and hence $Det(M^s_i)$ is a random instantiation [8] of $Det(\tilde{M}^s_i)$, which is identically not equal to 0. Hence, it follows from the classic theorem on checking polynomial identities [21, 23] that $\Pr(Det(M^s_i) = 0) \leq \frac{D}{p}$, where D is the degree of polynomial $Det(\tilde{M}^s_i)$ (in the variables whose random instantiation is used to obtain S'_i from $\tilde{\rho}'$). Note that $D \leq d_S$. Since $J(S'_i)$ is linearly dependent on $\{J(S'_j)\}_{j \in I_i}$ only if $Det(M^s_1) = 0$, we have:

$$\Pr(\neg\mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \;\wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j) \leq \frac{d_S}{p}$$

$$\begin{aligned}
&\text{Thus, } \Pr(\mathcal{E}_I) \\
&\leq \prod_{i \in \{1,..,t\}-I} \Pr\Big(\neg\mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \;\wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j\Big) \\
&\leq \prod_{i \in \{1,..,t\}-I} \frac{d_S}{p} \\
&= \left(\frac{d_S}{p}\right)^{t-|I|}
\end{aligned}$$

[8]This makes use of the assumption that `SEval` function does not involve any random variables because otherwise the choice of the random variables in S'_i is already decided by the choices that occur in other states S'_j and hence in $\mathtt{Det}(\tilde{M}^s_i)$.

$$
\begin{aligned}
\Pr(\mathcal{E}) &= \sum_{I\subseteq\{1,..,t\},1\in I,|I|<m} \Pr(\mathcal{E}_I) \\
&\le \sum_{I\subseteq\{1,..,t\},1\in I,|I|<m} \left(\frac{d_S}{p}\right)^{t-|I|} \\
&\le \sum_{i=1}^{m-1} \binom{t-1}{i-1} \left(\frac{d_S}{p}\right)^{t-i} \\
&\le \sum_{i=1}^{m-1} \left(\frac{3(t-1)}{t-i}\right)^{t-i} \times \left(\frac{d_S}{p}\right)^{t-i} \\
&\le \sum_{i=1}^{m-1} \left(\frac{3t}{t-(m-1)} \times \frac{d_S}{p}\right)^{t-i} \\
&\le \frac{\alpha^{t-k_{\mathrm{v}}}}{1-\alpha}, \text{ where } \alpha = \frac{3d_S t}{p(t-k_{\mathrm{v}})}
\end{aligned}
$$

We now show that $\gamma_1'(S) \le \Pr(\mathcal{E})$. Suppose that event $\mathcal{E}$ does not occur. Consider some equivalence $e_1 = e_2$ that is not entailed by $\tilde{\rho}$ in some context C. We show that S also does not entail the equivalence $e_1 = e_2$ in context C. Let $e = \texttt{SEval}(e_1) - \texttt{SEval}(e_2)$. Since $\tilde{\rho}$ does not entail $e_1 = e_2$ in context C, there exists some concrete state ρ belonging to the vector space $\mathcal{U}(\tilde{\rho})$ such that ρ does not entail $e_1 = e_2$ in context C, or equivalently, $e[\rho][C] \neq 0$. Since event $\mathcal{E}$ does not occur, there exist $\alpha_1, \ldots, \alpha_t \in \mathbb{F}_p'$ such that $J(\rho) = \sum_{i=1}^{t} \alpha_i J(S_i)$. Since $\sum_{i=1}^{t} \alpha_i = 1$, we have $e[\rho][C] = \sum_{i=1}^{t} (\alpha_i e[S_i])[C] = \sum_{i=1}^{t} \alpha_i[C](e[S_i][C])$. This implies that there exists $1 \le i \le t$ such that $e[S_i][C] \neq 0$. Thus, S_i (and hence S) does not entail the equivalence $e_1 = e_2$ in context C. This completes the proof.

D. PROOF OF LEMMA 2

We first define the notion of a *basic* set of contexts.

Definition 11. *[Basic Set of Contexts] A set of contexts B for a procedure P is said to be* basic *when for all contexts C and all equivalences $e_1 = e_2$ (such that the variables that occur in e_1 and e_2 have mappings in C), if $\texttt{Eval}(e_1, C) \neq \texttt{Eval}(e_2, C)$ then there exists a context $C' \in B$ such that $\texttt{Eval}(e_1, C') \neq \texttt{Eval}(e_2, C')$ and $C' \Rightarrow C$. We denote such a context C' by $Basic_B(C, e_1 = e_2)$.*

A basic set of contexts has the following property.

Property 12. *Let B be a basic set of contexts for a procedure P. Suppose that a summary Y_P for a procedure P is sound for all contexts in B. Then Y_P is sound for all contexts for procedure P.*

Proof. Let C be any context and $e_1 = e_2$ be any equivalence such that

$$\neg(Holds(e_1 = e_2, Paths(Y_P), C))$$

This implies that there exists a path $a \in Paths(Y_P)$ such that

$$\texttt{Eval}(e_1^a, C) \neq \texttt{Eval}(e_2^a, C)$$

where $e_i^a = e_i[g_m/x_m]..[g_1/x_1]$, and a is the sequence of assignments $x_1 = g_1; \ldots; x_m = g_m$. Let C_a be $Basic_B(C, e_1^a = e_2^a)$. By definition of C_a, we have

$$\texttt{Eval}(e_1^a, C_a) \neq \texttt{Eval}(e_2^a, C_a)$$

and hence

$$\neg(Holds(e_1 = e_2, Paths(Y_P), C_a))$$

It now follows from the soundness of Y_P for context C_a that

$$\neg(Y_P \models_{C_a} e_1 = e_2)$$

Since $C_a \Rightarrow C$, we conclude that

$$\neg(Y_P \models_C e_1 = e_2)$$

□

Observe that the following set is a basic set of contexts for any procedure P.

$$B \quad = \quad \{\{y_1 = v_1, .., y_k = v_k\} \mid y_i \in I_P, c_i \in \mathbb{F}_p\}$$

where $v_i \equiv \texttt{SEval}(y_i)[c_i/y_i]$ and I_P denotes the set of input variables of procedure P. Note that $|B| = p^{k_{\mathrm{i}}}$. Hence, $N \le p^{k_{\mathrm{i}}}$.