Precise Interprocedural Analysis using Random Interpretation

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 intraprocedural analyses, and also extends naturally to efficient interprocedural analyses. There is no such natural extension known for deterministic algorithms. We present a general technique for extending any intraprocedural random interpreter to perform a context-sensitive interprocedural 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 relationships interprocedurally in a linear program. We also obtain the first polynomial-time randomized algorithm for precise interprocedural value numbering over a program with unary uninterpreted functions.

We present experimental evidence that quantifies the precision and relative speed of the analysis for discovering linear relationships along two dimensions: intraprocedural vs. interprocedural, 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 the existing 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*

General Terms

Algorithms, Theory, Verification

Keywords

Interprocedural Analysis, Random Interpretation, Randomized Algorithm, Linear Relationships, Uninterpreted Functions, Interprocedural Value Numbering

1. INTRODUCTION

A sound and complete program analysis is undecidable [11]. 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* [2], 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 [5, 6], at the price of degrading soundness from absolute certainty to guarantee with arbitrarily high probability.

Until now, random interpretation has been applied only to intraprocedural analysis. Precise interprocedural analysis is provably harder than intraprocedural analysis [15]. There is no general recipe for constructing a precise and efficient interprocedural analysis from just the corresponding intraprocedural analysis. The functional approach proposed by Sharir and Pnueli [20] 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 [16], even for some kind of infinite domains. They successfully applied their technique to detect linear constants interprocedurally [19]. 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 intraprocedural 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 interprocedural 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'05, January 12–14, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-830-X/05/0001 ...$5.00.

Our technique is based on the standard procedure summarization approach to interprocedural 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 7) and unary uninterpreted functions (Section 8), for which there exist intraprocedural random interpretation 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 interprocedural value numbering [1] over a program with unary uninterpreted function symbols.

In the process of describing the interprocedural randomized algorithms, we develop a generic framework for describing both intraprocedural and interprocedural randomized analyses. This framework generalizes previously published random interpreters [5, 6], 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 interprocedural 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 could be obtained. We also compare experimentally the randomized interprocedural linear relationship analysis with an intraprocedural version [5] and with a deterministic algorithm [19] 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 intraprocedural 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 interprocedural randomized analysis. We prove the correctness of this algorithm in Section 5 and discuss fixed-point computation and complexity of this algorithm in Section 6. We instantiate this generic algorithm to obtain an interprocedural algorithm for discovering linear relationships, and for value numbering in Section 7 and Section 8 respectively. In Section 9, we describe how to avoid large numbers that may arise during computation, so that arithmetic can be performed using finite number of bits. Section 10 describes our experiments.

2. RANDOM INTERPRETATION

In this section, we formalize the intraprocedural 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) and (c). In the assignment node, x denotes a program variable, and e is either some *expression* or ?. Non-deterministic assignments $x :=?$ represent a safe abstraction of statements (in the original source program) that our abstraction cannot handle precisely. We also abstract the conditional guards by treating them as non-deterministic.

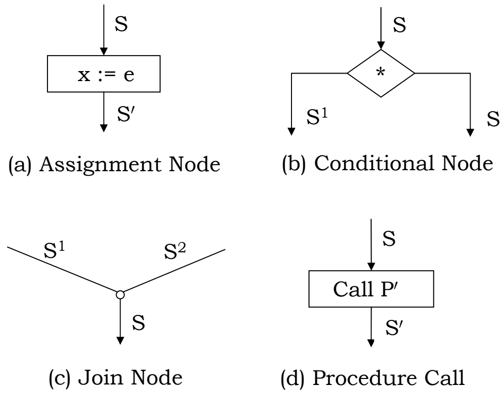


Figure 1: Flowchart nodes

A random interpreter executes a program on random inputs in a non-standard manner. An execution of a random interpreter computes a state ρ at each program point π. A state is a mapping from program variables to values v over some field $\mathbb{F}$.

A random interpreter processes ordinary assignments ($x := e$, where e is an expression) by updating the state with the value of the expression being assigned. 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. 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. The number of iterations `nrIters` required to reach a fixed point is abstraction specific.

In Section 2.4, we discuss how to verify or discover program equivalences from such random executions of a program. We also give a bound on the error probability in such an analysis. We give both generic and abstraction specific results. 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 some finite subset $\tilde{F}$ of $\mathbb{F}$. Let $q = |\tilde{F}|$. (In Section 9, we argue the need to perform arithmetic over a finite field, and hence choose $\mathbb{F} = \mathbb{Z}_q$, the field of integers modulo q, for some randomly chosen prime q. In that case $\tilde{F} = \mathbb{F}$.)

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:

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

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 [3]. 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,

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

For any polynomial P and any state ρ, we use the notation $[\![P]\!]\rho$ to denote the result of evaluation of polynomial P in state ρ. The affine join operator has the following useful properties. Let P and P' 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 q. Then,

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

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

A2. Soundness: If P and P' 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.

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

2.3 Eval Function

A random interpreter is equipped with an `Eval` function that takes an expression e and a state ρ and computes a field value. The `Eval` function plays the same role as an abstract interpreter's transfer function for an assignment operation. `Eval` is defined in terms of a symbolic counterpart `SEval` that translates an expression into a polynomial over the chosen field $\mathbb{F}$. This polynomial is linear in program variables, and may contain random variables as well, which stand for random field values chosen during the analysis. $\texttt{Eval}(e, \rho)$ is computed by replacing program variables in $\texttt{SEval}(e)$ with their values in state ρ, replacing the random variables with some random values, and then evaluating the result over the field $\mathbb{F}$.

`Eval` *function for Linear Arithmetic.* The random interpretation for linear arithmetic was described in a previous paper [5]. The following language describes the expressions in this domain. Here x refers to a variable and c refers to an arithmetic constant.

$$e ::= x \;\mid\; e_1 \pm e_2 \;\mid\; c \times e$$

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

`Eval` *function for Unary Uninterpreted Functions.* The random interpretation for uninterpreted functions was described in a previous paper [6]. We show here a simpler `SEval` function, for the case of unary uninterpreted functions. The following language describes the expressions in this domain. Here x refers to a variable and F refers to a unary uninterpreted function.

$$e ::= x \;\mid\; F(e)$$

The `SEval` function for this domain is as follows.

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

Here a_F and b_F are 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.

2.3.1 *Properties of the* `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.

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

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.

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

B3. Referential transparency:

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

B4. Linearity: The `SEval` function should be a polynomial that is linear in program variables.

Property B2 need not be satisfied if completeness is not an issue. This is of significance if the underlying theory is difficult to reason about, yet we are interested in a sound and partially complete reasoning for that theory. For example, the following `SEval` function for "bitwise or operator" ($||$) satisfies all the above properties except property B2.

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

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 of our paper can also be extended to prove relative completeness for a theory if the `SEval` function does not satisfy property B2.

2.4 Analysis and Error Probability

A random interpreter performs multiple executions of a program as described above, the result of which is a collection of multiple, say t, states at each program point. Let S^{π} denote the collection of t states at program point π. We can use these states 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 too.

2.4.1 *Verifying Equivalences*

The random interpreter declares two expressions e_1 and e_2 to be equivalent at program point π iff for all states $\rho \in S^{\pi}$, $\texttt{Eval}(e_1, \rho) = \texttt{Eval}(e_2, \rho)$. We denote this by $S^{\pi} \models e_1 = e_2$. 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 discovers all equivalences.

$$\pi \models e_1 = e_2 \Rightarrow S^{\pi} \models e_1 = e_2$$

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

$$\begin{aligned} \pi \not\models e_1 = e_2 \quad &\Rightarrow \quad \Pr(S^{\pi} \models e_1 = e_2) \leq \left(\frac{d}{q}\right)^t \\ d \quad &= \quad (n_j + \Delta) \times \texttt{nrIters} \end{aligned}$$

Here n_j denotes the number of join points in the program, and q denotes the size of the set from which the random interpreter chooses random values. For defining Δ, consider every sequence of assignments "$x_1 := e_1;\ \ldots;\ x_m := e_m$" along acyclic paths in the program. Δ is the maximum degree of any polynomial $\texttt{SEval}(e_i[e_{i-1}/x_{i-1}]\cdots[e_1/x_1])$, where $1 \leq i \leq m$. The arguments to $\texttt{SEval}$ are expressions computed on the path, expressed symbolically in terms of the input program variables. For example, Δ is 1 for the linear arithmetic abstraction. For the abstraction of uninterpreted functions, Δ is bounded above by the size of program (measured in terms of number of function applications). Intuitively, we add n_j 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 q to be greater than d and choosing t appropriately.

The proof of property C1 follows from a more general property (property D1 in Section 5) that states the completeness of an interprocedural random interpreter. The proof of property C2 is by induction on the number of steps performed by the random interpreter, and is similar to the proof of Theorem 2 (in Section 5) that states the soundness of an interprocedural random interpreter.

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. However, we do present a generic upper bound on the probability of error. For the case of linear arithmetic, the set of equivalences at a program point π can be described by a *basis* (in the linear algebraic sense) of the set of linear relationships that are true at point π. Such a basis can be mined from S^{π} by solving a set of simultaneous linear equations. For the case of uninterpreted functions, the set of all equivalences among program expressions can be described succinctly using an equivalence value graph [7]. This equivalence value graph can be built by first constructing the value flow graph [13] G of the program and then assigning t values $V_1(m), \ldots, V_t(m)$ to each node m in G as follows. A node m in a value flow graph G is either a variable x, or an F-node $F(m_1)$, or a ϕ-node $\phi(m_1, m_2)$ for some nodes m_1, m_2 in G. We use S_i to denote the i^{th} state in sample S.

$$\begin{aligned} V_i(x) &= S_i(x) \\ V_i(F(m)) &= \texttt{Eval}(F(V_i(m))), S_i) \\ V_i(\phi^j(m_1, m_2)) &= \phi_{w_i^j}(V_i(m_1), V_i(m_2)) \end{aligned}$$

Here w_i^j denotes the i^{th} random weight chosen for the j^{th} phi function ϕ^j in the value flow graph G. The nodes with equal values are merged, and the resulting graph reflects all equivalences among program expressions.

The following theorem states a generic upper bound on the probability that the random interpreter discovers any false equivalence at a given program point.

THEOREM 1. *For any program point π,*

$$\Pr(\exists e_1, e_2 : \pi \not\models e_1 = e_2 \ \wedge\ S_{\pi} \models e_1 = e_2) \leq \left(\frac{dk_v}{q}\right)^{\lfloor t/k_v \rfloor}$$

where d is as described in Section 2.4.1 and k_v is the maximum number of program variables visible at any program point.

The proof of Theorem 1 follows from a more general theorem (Theorem 2 in Section 5) that we prove for analyzing the correctness of an interprocedural random interpreter. Note that we must choose t greater than k_v, and q greater than dk_v.

For specific abstractions, it may be possible to prove better bounds on the error probability. For example, for the theory of linear arithmetic, it can be shown that the error probability is bounded above by $\left(\frac{d\beta}{q}\right)^{t-k_v}$, where $\beta = \frac{te}{t-k_v}$, e being the Euler constant $2.718\cdots$. For the theory of unary uninterpreted functions, it can be shown that the error probability is bounded above by $k_v^2 t^2 \left(\frac{d}{q}\right)^{t-2}$. This latter bound implies that a small error probability can be achieved with a value of t slightly greater than 2, as opposed to the generic bound, which requires t to be greater than k_v for achieving a low error probability. The proofs of these specific bounds follow from more general theorems that we state in Section 7 (Theorem 8) and Section 8 (Theorem 10) for analyzing the error probability of the interprocedural analysis for these abstractions.

This completes the description of the generic framework for random interpretation. We show an example next, and then we proceed to extend the framework to describe interprocedural analyses.

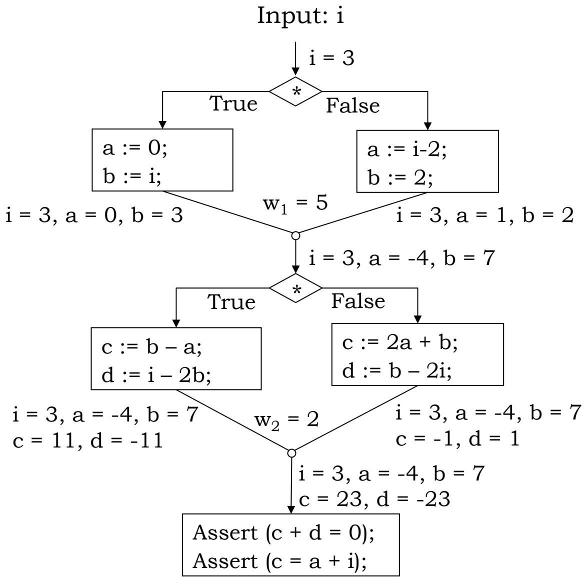


Figure 2: A code fragment with four paths. Of the two equations asserted at the end the first one holds on all paths but the second one holds only on three paths. The numbers shown next to each edge represent values of variables in a random interpretation.

2.5 Example

In this section, we illustrate the random interpretation scheme for discovering linear relationships among variables in a program 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 this procedure in isolation of the places it is used (i.e., intraprocedural 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 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 $\mathtt{i}$ and then executes the assignment statements on both sides of the conditional using the $\mathtt{Eval}$ 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 $\mathtt{w}_1 = 5$. 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 $\mathtt{a} + \mathtt{b} = \mathtt{i}$, 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 $\mathtt{w}_2 = 2$. We can then verify easily 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 invariant. Note that choosing $\mathtt{w}$ 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 $\mathtt{w}$ (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 $\mathtt{i}$, $\mathtt{w}_1$ and $\mathtt{w}_2$ that will make it conclude that both assertions hold (e.g., $\mathtt{i} = 2$, or $\mathtt{w}_1 = 1$).

In an interprocedural setting, the situation is more complicated. If this procedure is called only with input argument $\mathtt{i} = 2$, then both assertions hold, and the analysis is expected to infer that. One can also check that if the random interpreter chooses $\mathtt{i} = 2$, then it is able to verify both the assertions, for any choice of $\mathtt{w}_1$ and $\mathtt{w}_2$. We look next at what changes are necessary to extend the analysis to be interprocedural.

3. INFORMAL DESCRIPTION OF THE ALGORITHM

Our algorithm is based on the standard summary-based approach to interprocedural 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.

In this section, we briefly explain the two main ideas behind our summary computation technique that can be used to perform a precise interprocedural analysis using a precise intraprocedural random interpreter.

3.1 Random Symbolic Run

Intraprocedural 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 code fragment in Figure 2. The second assertion at the end of the code fragment is true in the context $\mathtt{i} = 2$, but this conditional fact is not captured by the random state at the end of the code fragment.

The first main idea of the algorithm is 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 polynomials in terms 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 code fragment from before, shown now in Figure 3. Note that the symbolic random state at the end of the code fragment (correctly) does not satisfy the second assertion. However, in a context where $\mathtt{i} = 2$, the state does satisfy $\mathtt{c} = \mathtt{a} + \mathtt{i}$ since $\mathtt{c}$ evaluates to 2 and $\mathtt{a}$ 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. In contrast, there seems to be no obvious way to extend an arbitrary

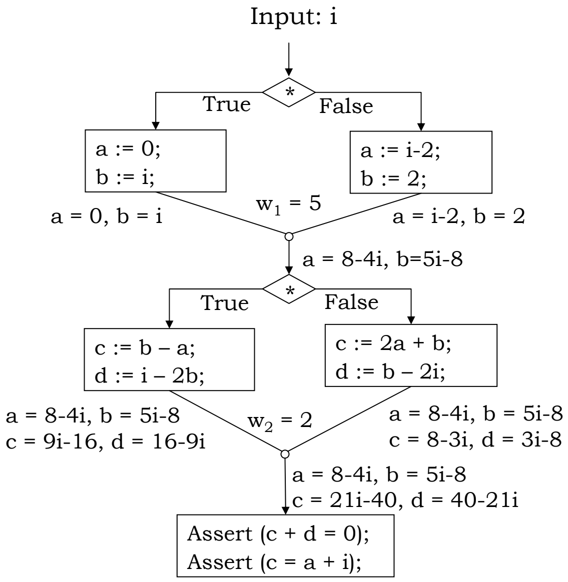


Figure 3: An illustration of symbolic random interpretation on the code fragment shown in Figure 2. Note that the second assertion is true in the context $\texttt{i} = 2$**, and the symbolic random interpreter verifies it.**

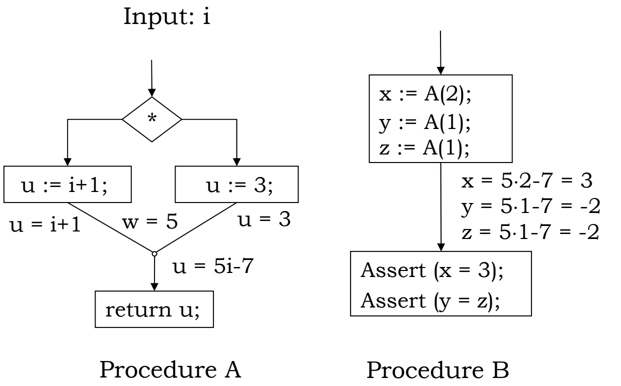


Figure 4: A program with 2 procedures. The first assertion at the end of procedure B is true, while the second assertion is not true, because procedure A may run different branches in different runs. This figure demonstrates that use of just one random symbolic run is unsound.

intraprocedural abstract interpretation scheme to perform a context-sensitive interprocedural analysis. This is the case, for example, for the intraprocedural abstract interpretation for linear arithmetic that was first described by Karr [9] in 1976. An interprocedural abstract interpretation for linear arithmetic was described by Müller-Olm and Seidl [14] only recently in 2004.

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 since the non-deterministic conditional (which arises as a result of our conservative abstraction of not modeling the conditional guards) in procedure `A` can evaluate differently in the two calls to procedure `A`, 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 naive way to generate n fresh runs for a procedure P is to execute n 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 F_i calls procedure F_{i+1} two times. To generate a run for F_0, we need 2 fresh runs for F_1, which are obtained using 4 fresh runs for F_2, and so on.

The second main idea in our algorithm is that we can generate the equivalent of t fresh runs for a procedure P if t fresh runs are available for each of the procedures that P calls, for some parameter t that depends on the underlying abstraction. This idea relies on the fact that an affine combination 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 generate 2 fresh runs for procedure `B`, we need 6 fresh runs for procedure `A`. The figure shows computation of 6 fresh runs from the 2 runs for procedure `A`. The first call to procedure `A` uses the first two 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. INTERPROCEDURAL RANDOM INTERPRETATION

We now describe the precise interprocedural randomized algorithm, as 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. We first describe our program model.

4.1 Preliminaries

A program is a set of procedures, each with one entry and one exit node. We assume that the flowchart representation of a procedure consists of nodes of the kind shown in Figure 1. For simplicity, we assume that the inputs and outputs

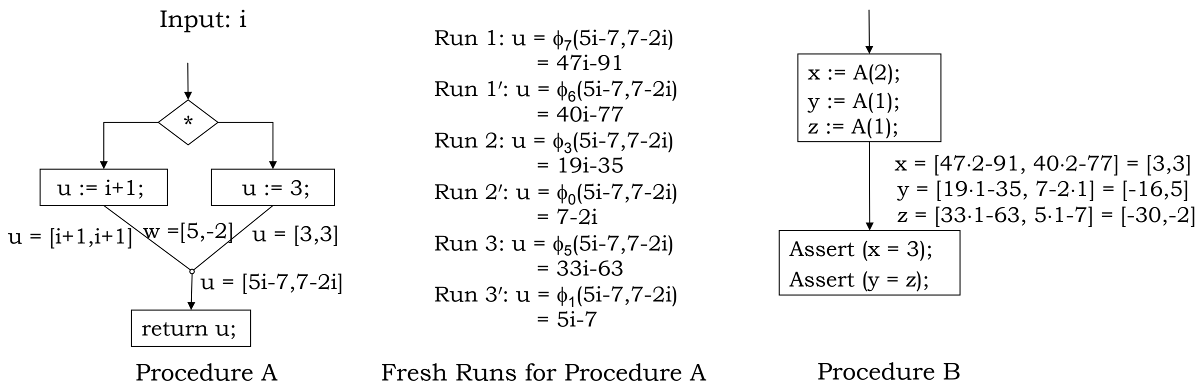


Figure 5: A program with two procedures, which is also shown in Figure 4. This figure illustrates the random interpretation technique of computing multiple random symbolic runs (in this example, 2) for a procedure, and using them to generate a fresh random symbolic run for every call to that procedure. 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.

of a procedure are passed as global variables. We use the following notation:

- I_P: Set of input variables for procedure P.
- O_P: Set of output variables for procedure P.
- k_I: Maximum number of input variables for any procedure.
- k_o: Maximum number of output variables for any procedure.
- k_v: Maximum number of visible variables at any program point.
- n: Number of program points.
- f: Fraction of procedure calls to the number of program points.
- r: Ratio of number of ϕ assignments (in the SSA version of the program) to the number of program points.

Note that the set I_P includes the set of all global variables read by procedure P directly as well as the set $I_{P'}$ for any procedure P' called by P. Similarly for set O_P.

4.2 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 is a mapping from output variables in O_P to random symbolic values, which are linear expressions in terms of the input variables I_P of procedure P. The value of t is abstraction dependent. (For linear arithmetic, the value of $t = k_I + k_v + c$ suffices, for some small constant c. For unary uninterpreted functions, the value of $t = 4 + c$ suffices.)

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 symbolic (random) 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 starts by initializing to $\perp$ the summaries of all procedures, and the samples at all program points except at procedure entry points. The samples at entry points are initialized to $S_i(x) = x$ for every input variable x and $1 \leq i \leq t$. 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 four kinds of flowchart nodes are described below. After the random interpreter is done interpreting a procedure, the summary is simply the projection of the sample (or the t states) at the end of the procedure to the output variables of the procedure. The random interpreter interprets each flowchart node as follows.

Assignment Node. See Figure 1(a). Let S be the sample before an assignment node $x := e$. If S is $\perp$, then the sample S' is defined to be $\perp$. Otherwise, the random interpreter computes S' by executing the assignment statement in sample S as follows.

$$S'_i(x) \;=\; \begin{cases} \texttt{rand()} & \text{if } e \text{ is } ? \\ \texttt{Eval}(e, S_i) & \text{otherwise} \end{cases}$$

$$S'_i(y) \;=\; S_i(y), \text{for all variables } y \text{ other than } x.$$

Here `rand()` denotes a fresh random value.

Conditional Node. See Figure 1(b). Let S be the sample before a conditional node. The samples S^1 and S^2 after the conditional node are simply defined to be S. This reflects the fact that we abstract away the conditional guards.

Join Node. See Figure 1(c). Let S^1 and S^2 be the two samples immediately before a join point. If S^1 is $\perp$, then the sample S after the join point is defined to be S^2. Similarly, if S^2 is $\perp$, then $S = S^1$. Otherwise, the random interpreter selects t random weights $w_1, \ldots, w_t$ to perform a join operation. The join operation involves taking a random affine combination of the corresponding states in the two samples S^1 and S^2 immediately before the join point.

$$S_i = \phi_{w_i}(S_i^1, S_i^2)$$

Procedure Call. See Figure 1(d). Let S be the sample before a call to procedure P', whose input (global) variables are $i_1, \ldots, i_k$. If S 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.

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

The effect of a call to procedure P' is to update the values of the variables $O_{P'}$ that are written to by procedure P'. The random interpreter models this effect by updating the values of the variables O_P using the fresh random runs Y_i (computed above) as follows.

$$S_i'(x) = \begin{cases} Y_i(x)[S_i(i_1)/i_1, \ldots, S_i(i_k)/i_k] & \text{if } x \in O_{P'} \\ S_i(x) & \text{otherwise} \end{cases}$$

4.3 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. The samples are computed for each program point from the samples at the preceding program points in exactly the same manner as described in Section 4.2. However, there are two main differences. First, the states are mapping of program variables to constants, i.e., elements of the field $\mathbb{F}$. Second, the sample S at the entry point of a procedure P is obtained as an 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 variable x,

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

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 (involving input variables of procedure P) that hold in all calls to procedure P.

4.4 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 [3] 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 interprocedural random interpretation. We sketch next the proof of correctness.

5. CORRECTNESS OF THE ALGORITHM

A sample S^π computed by the random interpreter at a program point π has the following properties. We use the term *input context*, or simply, context, to denote any set of equivalences between program expressions involving only the input variables.)

D1. Completeness: In all input contexts, S^π entails all equivalences that hold at π along the paths analyzed by the random interpreter.

D2. Soundness: With high probability, in all input contexts, S^π entails only the equivalences that hold at π along the paths analyzed by the random interpreter. The error probability $\gamma(t)$ (assuming that the preceding samples satisfy property D2) is bounded above by

$$\gamma(t) \le q^{k_I} \left(\frac{dk_v}{q}\right)^{\lfloor t/k_v \rfloor}$$

where $d = n_j + n_c + \Delta$. Here n_j refers to the number of join points and n_c refers to the number of procedure calls along any path analyzed by the random interpreter. Δ refers to the maximum degree of $\texttt{SEval}(e)$ for any expression e computed by the random interpreter (and expressed in terms of the input variables of the program) along any path analyzed by it. [1]

We briefly describe the proof technique used to prove these properties. Detailed proof can be found in the full version of the paper [8].

First 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. 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 both sound and complete, even though it might be computationally expensive. The proof of this fact is by induction on the number of steps performed by the random interpreter.

Property D1 follows directly from the fact that the t states that make up a sample in the random interpreter can all be obtained by instantiating the random variables in the fully-symbolic state.

[1] Δ is 1 for the linear arithmetic abstraction. For the abstraction of uninterpreted functions, Δ is bounded above by the maximum number of function applications processed by the random interpreter along any path analyzed by it.

We now prove property D2 in two steps. (For practical reasons, we perform arithmetic over the finite field $\mathbb{Z}_q$, which is the field of integers modulo q, for some randomly chosen prime q. This is described in more detail in Section 9. Hence, we compute and show that the value of $\gamma(t)$ is small under this assumption. It is possible to do the proof without this assumption, but the proof is more complicated.)

Step 1. 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 $\tilde{\rho}$ of degree d in *a given context.* The following theorem specifies a bound on this error probability, which we refer to as the discovery factor $D(d,t)$. It is possible to prove better bounds for specific abstractions.

THEOREM 2. $D(d,t) \le \left(\frac{dk_v}{q}\right)^{\lfloor t/k_v \rfloor}$.

The proof of this theorem is in the full version of the paper [8]. Note that this theorem also provides a bound on the error probability in the process of discovering equivalences for the intraprocedural analysis (Theorem 1).

Step 2. 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. It is possible to prove better bounds for specific abstractions.

THEOREM 3. $N \le q^{k_I}$.

The proof of this theorem is in the full version of the paper [8].

The probability that a sample S^π is not sound in any of the contexts is bounded above by the probability that S^π is not sound in any one given context multiplied by the size of any basic set of contexts. Thus, the error probability $\gamma(t)$ mentioned in property D2 is bounded above by $D(d,t) \times N$.

6. FIXED POINT COMPUTATION AND COMPLEXITY

The notion of loop that we consider for fixed point computation is that of a 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 [16]. This directed graph representation consists of a collection of flowgraphs, one for each procedure in the program, with the addition of some new edges. For every edge to a call node, say from node n_1 to call node n_2 with the call being to procedure P, we add two new edges: one from node n_1 to start node of procedure P, and the other from exit node of procedure P to node n_2. Now consider the DAG D of SCCs of this directed graph representation of the program. Note that an SCC in DAG D may contain nodes of more than one procedure [2] (in which case it contains all nodes of those procedures).

In both phase 1 and phase 2, a random interpreter processes all SCCs in the DAG D 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 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.

To prove a bound on H_1, we first observe that there exists a concise deterministic representation for the sets of equivalences, one for each context, that can arise at any program point. The set of all equivalences in one given context can usually be succinctly described. For example, the set of all linear equivalences can be represented by a basis [9]. For the domain of uninterpreted functions, the set of all Herbrand equivalences can be described by a value graph [7]. However, it it challenging to represent these sets concisely for all contexts, which are potentially infinite in number. We illustrate such a representation by means of an example.

Consider the following program fragment.

```
if (*) then { x := a; }
       else { x := 3; }
if (*) then { y := b; z := c; }
       else { y := 2; z := 2; }
```

The sets of equivalences for all contexts can be represented by the pair $\langle E, L\rangle$, where E is a set of relationships involving program variables and a set of “join” variables λ,

$$\begin{array}{rcl} x - 3 + \lambda_1(3-a) & = & 0 \\ y - 2 + \lambda_2(2-b) & = & 0 \\ z - 2 + \lambda_3(2-c) & = & 0 \end{array}$$

and L is the following set of relationships among λ's.

$$\lambda_2 = \lambda_3$$

The above set of equations capture all equivalences among the program expressions in all contexts. For example, note that these equations imply that $x = 3$ in the context $\{a = 3\}$, or $y = z$ in the context $\{b = c\}$.

Theorem 4 stated below describes more precisely this representation.

THEOREM 4. *At any program point π, the sets of equivalences for all contexts can be described by a pair $\langle E, L\rangle$, where E is a set of $\ell \le k_v$ equations of the form:*

$$p_i + \sum_{j=1}^{a_i} \lambda_i^j p_i^j = 0 \qquad 1 \le i \le \ell$$

with the following properties:

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

E1. For all $1 \le i \le \ell$, $p_i^1 = 0, \ldots, p_i^{a_i} = 0$ are independent equations that are linear over the input variables of the enclosing procedure, and $a_i \le k_I$.

E2. $p_1 = 0, \ldots, p_\ell = 0$ are independent equations that are linear over the program variables visible at π.

and L is a set of linear relationships among λ_i^j's.

The proof of Theorem 4 is in the full version of the paper [8]. It must be pointed out that the representation described by the above theorem is only of academic interest; it has been introduced for proving a bound on H_1. It is not clear how to construct such a representation efficiently.

We now state and prove the theorem that bounds H_1.

Theorem 5. *$H_1 \le (2k_I + 1)k_v + 1$.*

Proof. Consider the representation $\langle E, L\rangle$ of the sets of equivalences for all contexts (as described in Theorem 4) that can arise at a program point π. For any pair $\langle E, L\rangle$, we define measure $M(\langle E, L\rangle)$ to be the sum of the size of L and the sum of integers in the set $\{1+k_I-a_i \mid p_i + \sum_{j=1}^{a_i} p_i^j \in E\}$.

Now suppose that at some program point in an SCC, the sets of equivalences for all contexts are described by $\langle E_1, L_1\rangle$ in some iteration and by $\langle E_2, L_2\rangle$ in a later iteration. Then it must be the case that for all contexts C, the set of equivalences described by $\langle E_2, L_2\rangle$ in context C is implied by the set of equivalences described by $\langle E_1, L_1\rangle$ in context C. Using this, it can be shown that if $\langle E_1, L_1\rangle$ and $\langle E_2, L_2\rangle$ do not represent the same sets of equivalences, then $M(\langle E_2, L_2\rangle) < M(\langle E_1, L_1\rangle)$. Note that $M(\langle E, L\rangle)$ is non-negative with a maximum value of $(2k_I + 1)k_v$. Hence, $H \le (2k_I + 1)k_v + 1$. □

The following theorem states a bound on H_2.

Theorem 6. *$H_2 \le k_v + 1$.*

The proof of the above theorem follows from the observation that the set of equivalences at any program point in phase 2 can be represented by a set of at most k_v equations that are linear over the program variables visible at that point.

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. Hence, the random interpreter blindly goes around each SCC as many times as is sufficient for reaching fixed-point. Note that the edges in an SCC S can be decomposed into a DAG D_S and a set of back-edges. If a random interpreter processes the nodes inside an SCC S in a top-down manner according to their ordering in the DAG D_S, then it needs to process each SCC for $H_1(b+1)$ steps in phase 1 and for $H_2(b+1)$ steps in phase 2, where b denotes the number of back-edges in that SCC. Note that this guarantees that a fixed point has been reached. Since b is typically very small, as experiments also suggest, we ignore it in our further discussion.

We now state the time complexity of the random interpreter as a function of the error probability. We assume a unit time complexity for our arithmetic operations since we perform arithmetic over a small finite field. However, performing arithmetic over a finite field may lead to some additional error probability, which we analyze in Section 9.

Theorem 7. *For any constant c, if we choose $t = k_I k_v + c$, then the random interpreter runs in time $O(nk_v k_I^2 t(1 + r + ftk_o))$ and has an error probability of $O((\frac{1}{q})^c)$.*

Proof. The total number of samples computed by the random interpreter is $n_s = n(H_1 + H_2)$. We assume the use of optimization described in Section 4.4. The computation of a sample across an assignment node takes $O(k_I t)$ time (assuming that the expression assigned is of constant size, and the `Eval` function takes $O(k_I)$ time). The processing of a join node takes $O(pk_I t)$ time, where p is the number of ϕ assignments at the join node after SSA conversion. The cost of executing a procedure call is $O(k_I t^2 k_o)$. Hence, the total cost of computing all samples is $O(n_s k_I t(1 + r + ftk_o))$.

Each sample can be unsound with an error probability of $\gamma(t)$, which is defined in property D2 in Section 5. Hence, the total error probability is bounded above by $n_s\gamma(t)$. Assuming that q is significantly greater than the various parameters d, k_I, k_v, n, the error probability decreases exponentially with c, where $c = t - k_I k_v$. □

If we perform arithmetic using 32 bit numbers, then $q \approx 2^{32}$ and the error probability can be made arbitrarily small with even a small value of c. The ratio r is bounded above by k_v. However, it has been reported [3] that r typically varies between 0.3 to 2.8 irrespective of program size. If we regard k_I and k_o to be constants, since they denote the size of the interface between procedure boundaries and are supposedly small, then the complexity of our algorithm reduces to $O(nk_v^2)$, which is linear in the size of the program and quadratic in the maximum number of visible program variables at any program point.

7. APPLICATION TO DISCOVERING LINEAR RELATIONSHIPS

In this section, we apply the generic interprocedural analysis developed in the earlier sections to the abstraction of linear arithmetic. For this abstraction, it is possible to prove a better bound on one of the parameters $D(d,t)$, which controls the complexity of the generic algorithm.

Theorem 8. *$D(d,t) \le \left(\frac{d}{q}\frac{te}{t-k_v}\right)^{t-k_v}$. Here e denotes the Euler constant $2.718\cdots$.*

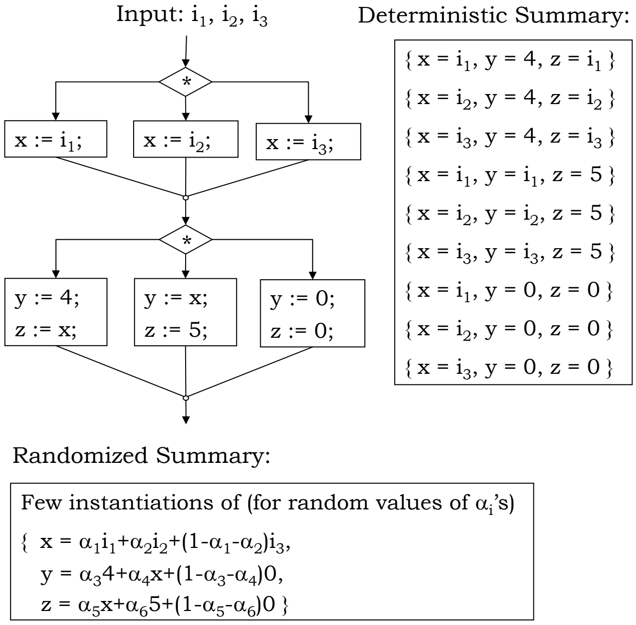
The proof of Theorem 8 is in the full version of the paper [8].

Theorem 8 implies the following complexity bound, which is better than the generic bound.

Theorem 9. *For any constant c, if we choose $t = k_v + k_I + c$, then the random interpreter runs in time $O(nk_v k_I^2 t(1 + r + ftk_o))$ and has an error probability of $O((\frac{1}{q})^c)$.*

It must be mentioned that the above bound is a conservative bound in terms of the constraint on t. Experiments discussed in Section 10 suggest that even $t = 3$ does not yield any error in practice.

Related Work. Recently, Muller-Olm and Seidl gave a deterministic algorithm (MOS) that discovers all linear relationships in programs that have been abstracted using nondeterministic conditionals [14]. The MOS algorithm is also based on computing summaries of procedures. However, their summaries are deterministic and consist of linearly



Deterministic Summary:

{ $x = i_1$, $y = 4$, $z = i_1$ }
{ $x = i_2$, $y = 4$, $z = i_2$ }
{ $x = i_3$, $y = 4$, $z = i_3$ }
{ $x = i_1$, $y = i_1$, $z = 5$ }
{ $x = i_2$, $y = i_2$, $z = 5$ }
{ $x = i_3$, $y = i_3$, $z = 5$ }
{ $x = i_1$, $y = 0$, $z = 0$ }
{ $x = i_2$, $y = 0$, $z = 0$ }
{ $x = i_3$, $y = 0$, $z = 0$ }

Randomized Summary:

Few instantiations of (for random values of α_i's)
{ $x = \alpha_1 i_1 + \alpha_2 i_2 + (1-\alpha_1-\alpha_2) i_3$,
$y = \alpha_3 4 + \alpha_4 x + (1-\alpha_3-\alpha_4) 0$,
$z = \alpha_5 x + \alpha_6 5 + (1-\alpha_5-\alpha_6) 0$ }

Figure 6: A code fragment that illustrates the difference between the deterministic summaries computed by MOS algorithm and the randomized summaries computed by our algorithm.

independent runs of the program. The program shown in Figure 6 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_v^2 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 $k_v + k_I$ runs are required while experiments suggest that even 3 runs are good enough.

The authors have proved a complexity of $O(nk_v^8)$ for the MOS algorithm assuming a unit cost measure for arithmetic operations. However, it turns out that the arithmetic constants that arise in MOS algorithm may be so huge that $\Omega(2^n)$ bits for required for representing constants, and hence $\Omega(2^n)$ time is required for performing a single arithmetic operation. Thus, the complexity of MOS algorithm is exponential in n. Program P_m shown in Figure 7 in Section 9 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, as described in Section 9. 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 interprocedurally in a program [19]. Their analysis only considers 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 relationships (not just constants).

The first intraprocedural analysis for discovering all linear relationships was given by Karr much earlier in 1976 [9]. The fact that it took several years to obtain an interprocedural analysis to discover all linear relationships in linear programs demonstrates the complexity of interprocedural analysis.

8. APPLICATION TO VALUE NUMBERING

In this section, we discuss the application of our technique to discovering equivalences among expressions built from unary uninterpreted functions. This abstraction is very useful in modeling fields of data-structures and can be used to compute must-alias information. For this abstraction, it is possible to prove better bounds on the parameters $D(d,t)$ and N, which control the complexity of the generic algorithm.

Theorem 10. $D(d,t) \leq k_v^2 t^2 \left(\frac{d}{q}\right)^{t-2}$.

The proof of Theorem 10 is similar to the proof of Theorem 8 and is based on the observation that any equivalence in the theory of unary uninterpreted functions involves only 2 variables, rather than k_v variables.

Theorem 11. $N \leq k_v^2 q^2$.

The proof of Theorem 11 is given in the full version of the paper [8].

Theorem 10 and Theorem 11 imply the following complexity bound, which is better than the generic bound.

Theorem 12. *For any constant c, if we choose $t = 4 + c$, then the random interpreter runs in time $O(nk_v k_I^2 t(1 + r + ftk_o))$ and has an error probability of $O(\frac{1}{q^c})$.*

Related Work. There have been number of attempts at developing intraprocedural 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 [10, 17, 18]. Recently, we presented a randomized [6] as well as a deterministic [7] polynomial time complete algorithm for this problem.

We are not aware of any complete interprocedural algorithm for value numbering. Gil and Itai have characterized the complexity of a similar problem, that of type analysis of object oriented programs in an interprocedural setting [4].

9. ARITHMETIC IN A FINITE FIELD

We first illustrate the problem that arises from performing arithmetic over an infinite field. Consider the program shown in Figure 7. Any summary-based approach for discovering linear relationships will essentially compute the following summary for procedures F_m and G_m:

$$F_m(x) = 2^{2^m} x$$
$$G_m(x) = 2^{2^m} x$$

Note that representing the arithmetic constant 2^{2^m} using standard decimal notation requires $\Omega(2^m)$ digits.

```
F_0(x)   = { return 2x; }

F_i(x)   = { t := F_{i-1}(x); return F_{i-1}(t); }

G_0(x)   = { return 2x; }

G_i(x)   = { t := G_{i-1}(x); return G_{i-1}(t); }

Main() = { t_1  := F_m(0); t_2  := G_m(0);

                      Assert(t_1 = t_2); }
```

Figure 7: A program P_m for which any deterministic summary based interprocedural analysis would require $\Omega(2^m)$ space and time for manipulating arithmetic constants (assuming standard binary representation). The program P_m contains $2m+3$ procedures Main, F_i and G_i for every integer $i \in \{0,\ldots,m\}$. A randomized analysis does not have this problem.

The problem of big numbers can be avoided by performing computations over the finite field $\mathbb{Z}_q$, which is the field of integers modulo q, for some randomly chosen prime q, instead of working over the infinite field [12]. However, this may result in an additional error probability in the algorithm, if the soundness property (property B1) of the SEval function described in Section 2.3.1 breaks down. The soundness of SEval function for the abstraction of uninterpreted functions does not rely on the underlying field; hence performing arithmetic over a finite field in that case does not introduce any additional error probability. However, the soundness of SEval function for linear arithmetic abstraction breaks if arithmetic is performed over a finite field. We next describe the resulting additional error probability in that case for both discovering and verifying equivalences.

For the purpose of discovering equivalences, we need to run the algorithm described in Section 4 two times, each time with an independently chosen random prime, say q_1 and q_2. The equivalences discovered in the first run of the algorithm are verified in the second run of the algorithm. This is required because if there is an equivalence involving large constants, say $y = bx$ for some large constant b and variables x and y, then the first run of the algorithm will print out an spurious equivalence $y = (b \mod q_1)x$. The second run of the algorithm will (very likely) invalidate this spurious equivalence since the probability that $b \mod q_1 = b \mod q_2$ for randomly chosen large enough q_1 and q_2 is very small. However, note that this technique does not discover the equivalence $y = bx$; in general it does not discover equivalences that involve constants that are larger than the randomly chosen prime numbers. Perhaps this is the best that can be hoped from a polynomial time algorithm since the size of the constants in equivalences may be exponential in the size of the program, as illustrated by the example in Figure 7. The resulting additional error probability for discovering equivalences is given by the product of the number of independent equivalences discovered and the additional error probability for verifying an equivalence.

The additional error probability for verifying an equivalence is simply the error probability that two given distinct integers are equal modulo a random prime q. The following property states this error probability.

	Prime Used								
t	983			65003			268435399		
	E_1 [a]	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

[a] $E_i = ((m'_i - m_i)/m'_i) \times 100$
m'_1 = # of reported variable constants x=c
m'_2 = # of reported variable equivalences x=y
m'_3 = # of reported dependent induction variables
m_i = # of correct relationships of the corresponding kind

Table 1: The percentage E_i of incorrect relationships of different kinds discovered by the interprocedural random interpreter as a function of the number of runs and the randomly chosen primes on a collection of programs, which are listed in Table 2. The total number of correct relationships discovered were $m_1 = 3442$ constants, $m_2 = 4302$ variable equivalences, and $m_3 = 50$ dependent induction variables.

Property 13. *The probability that two distinct m bit integers are equal modulo a randomly chosen prime less than U is at most $\frac{1}{u}$ for $U > 2um\log(um)$.*

Theoretically, the size of constants that can arise in the equivalences can be $O(2^n)$, as illustrated by the example in Figure 7. Hence, theoretically, to make the error probability arbitrarily small, we need to work with primes whose size is $O(n)$. However, in practice, the size of all equivalences is bounded by a small constant, and hence working with a small constant sized prime yields a low error probability. Our experiments suggest that a 32 bit prime is good enough in practice, as it did not yield any false equivalence.

10. EXPERIMENTS

In the earlier part of this paper we have expanded the body of theoretical evidence that randomized algorithms may have certain advantages, such as simpler implementations and better computational complexity, over deterministic ones. In this paper, for the first time, we investigate experimentally the behavior of these algorithms. The goal of these experiments are threefold: (1) attempt to measure experimentally the soundness probability and how it varies with certain parameters of the algorithm, (2) measure the running time and effectiveness of the interprocedural version of the algorithm, as compared with the intraprocedural version, and (3) perform a similar comparison with a deterministic interprocedural 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 [19]. We measured running time using enough repetitions to avoid timer resolution errors.

We have implemented the interprocedural 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

Program	Lines	Random Inter.						Random Intra.				Determ. Inter.	
		in	x=c	x=y	dep	Time$_2$	Time$_3$	$\Delta_{x=c}$	$\Delta_{x=y}$	Δ_{dep}	Speedup$_3$	Δ_{in}	Speedup$_2$
go	28622	63	1700	796	6	47.3	70.4	170	260	3	107	17	1.9
ijpeg	28102	31	825	851	12	3.8	5.7	34	1	9	24	3	2.3
li	22677	53	392	2283	9	34.0	51.4	160	1764	6	756	20	1.3
gzip	7760	49	525	372	2	2.0	3.05	200	12	1	39	6	2.0
bj	2109	0	117	9	0	1.2	1.8	0	0	0	11	0	2.3
linpackc	1850	14	86	16	1	0.07	0.11	17	1	1	9	0	1.8
sim	1555	3	117	296	0	0.35	0.54	3	11	0	22	0	1.7
whets	1275	9	80	2	6	0.03	0.05	17	1	0	9	0	1.5
flops	1151	0	52	4	4	0.02	0.03	0	0	0	22	0	2.0

Table 2: The column "in" shows the number of linear relationships involving the input variables at the start of procedures. The column "x=c" shows the number of variables equal to constant values. The column "x=y" shows the number of variable equivalences. The column "dep" shows dependent loop induction variables. The "Time$_t$" columns show running time (in seconds) for $t = 2$ and $t = 3$. The columns labeled "Δ" show how many *fewer* relationships of a certain kind are found with respect to the interprocedural randomized algorithm. The columns labeled "Speedup$_t$" columns show how many times faster is each algorithm with respect to the interprocedural randomized one with a given value of t.

increase in number of runs and the size of the prime number we use for modular arithmetic. The last two parameters have a direct impact on the running time.[3]

We first ran the interprocedural randomized analysis on our suite of programs, using a variable number of runs, and three 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).[4] 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, for which the theory requires $t > k_v$, yet in practice we do not notice any errors for $t \geq 4$. Similarly, our theoretical probability of error 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. [5] In fact, for the simpler problem of discovering equalities, 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 interprocedural analysis with that of randomized intraprocedural analysis and deterministic interprocedural analysis.

The first set of columns in Table 2 show the results of the interprocedural 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 intraprocedural randomized analysis, and how much faster that analysis is, when compared to the interprocedural one. The intraprocedural 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 are similar to those reported in [19]. Whether the additional information collected by the interprocedural analysis is worth the extra implementation and compile-time cost will depend on how the relationships are used. For compiler optimization it is likely that intraprocedural results are good enough, but perhaps for applications such as program verification the extra cost might be worth paying.

Finally, we wanted to compare our interprocedural algorithm with similar deterministic algorithms. We have implemented and experimented with the SRH [19] algorithm, and the results are shown in the third set of columns in Table 2. SRH is weaker than our algorithm, in that it searches only for equalities with constants. 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_{in}), 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 [14] algorithm, which is the most powerful deterministic algorithm known for this problem. 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

[3] For larger primes, the arithmetic operations cannot be inlined anymore.

[4] We found many more linear dependencies, but report only the induction variable ones because those have a clear use in compiler optimization.

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

points. Another source of bottleneck may be the complexity of manipulating large constants that may arise during the 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 relationships 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 interprocedural 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 intraprocedural random interpretations with symbolic values for input variables. Using this observation, we have obtained interprocedural 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.

Acknowledgments

We thank the anonymous reviewers for their useful feedback on a draft of this paper.

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] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. Technical Report UCB//CSD-04-1353, University of California, Berkeley, 2004.

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

[10] 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.

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

[12] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.