Efficient Directionless Weakest Preconditions

David Brumley, Ivan Jager

February 2, 2010

(revised July 14, 2010)

CMU-CyLab-10-002

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Efficient Directionless Weakest Preconditions

David Brumley Ivan Jager
Carnegie Mellon University
{dbrumley,aij}@cmu.edu

Abstract

Verification condition (VC) generation is a fundamental part of many program analysis and applications, including proving program correctness, automatic test case generation, and proof carrying code.

One might imagine VC applications would use the theoretically most appealing VC generation algorithm. This is often not the case. The most theoretically appealing algorithms are based upon weakest preconditions, and generate VCs at most $O(M^2)$ for M program statements, however they process statements from last to first. In practice, many application domains choose forward symbolic execution (FSE), which generates predicates $O(2^M)$ in size. FSE is chosen because it builds VCs in execution order, which allows a number of pragmatic optimizations.

We reconcile the differences between FSE and WP by proposing a new directionless weakest precondition that can be run in both the forward and backward direction. Our algorithm provides the more attractive $O(M^2)$ VC generation time and predicate size while allowing VC generation in execution order, which is what makes FSE attractive in practice. Thus, our algorithm can act as a "drop-in" replacement for either algorithm. We provide proofs of correctness, size, and generation time. We also show a correspondence between FSE and WP for deterministic programs, and perhaps surprisingly that they are not equivalent for the full class of GCL programs.

1. Introduction

A fundamental primitive in program analysis is the ability to automatically generate a formula that captures the semantics of a program fragment and its correctness properties. Such a formula is called a *verification condition* (VC). VCs allow us to reduce the problem of reasoning about programs to reasoning in logic.

Two of the most common algorithms for computing VCs from a program are weakest preconditions (WP) and forward symbolic execution (FSE). Although both approaches generate a VC from a program, the algorithms themselves are quite different, and previously it has not been shown whether or not the actual VCs generated are equivalent. Furthermore, the two approaches currently enjoy different properties and are often used in very different scenarios.

FSE algorithms build up a VC by "executing" the program in a symbolic interpreter. When the interpreter encounters a statement requiring user input, it returns a fresh symbolic variable (instead of a concrete value). Subsequent execution builds up an expression, f, in terms of the symbolic inputs for each executed path. Just like with concrete execution, each instance of the symbolic interpreter executes exactly one path, and thus creates one formula per path. Conditional statements cause the current interpreter instance to "fork" a new interpreter instance for each code branch, producing formulas $f_1, f_2, ..., f_n$ for each executed code path. The overall VC semantics is $f_1 \vee f_2 \vee ... \vee f_n$ for n code paths. [1]

Weakest preconditions algorithms build up a VC in the backward direction (i.e. from the last statement to first) by applying an appropriate predicate transformer for each encountered statement type. Roughly speaking, a *predicate transformer* takes the current VC f and a program statement s and produces a new VC f' such that if f' is true before executing, f will be true after executing s. With both FSE and WP, the final VC is satisfiable for inputs that would cause the program to terminate, optionally also satisfying an additional postcondition.

Nevertheless, there are many practical and theoretical differences between work using WP and FSE that impact their applicability, cost, and scalability. These factors have caused research using FSE and WP-style VC generation to progress mostly independently. For example, papers employing FSE typically have little or no mention of WP or its benefits (e.g., [6, 7, 14, 18, 19, 27]), and vice-versa. Specific differences include:

VC Size. VCs produced using the Flanagan and Saxe algorithm [15] are $O(M^2)$ in size, where M is the number of program statements. FSE-style algorithms produce a VC exponential in the number of statements. The smaller WP VCs save more than just bits on disk; previous work has demonstrated that they can be solved faster by typical decision procedures in practice [15, 21].

Path Selection. FSE research enjoys a rich set of algorithms for selecting paths of interest when generating VCs, such as concolic execution [14, 27], bounded depth-first search [6, 7], and generational search [19]. These path selection strategies make little sense for WP given that WP algorithms do not build VCs in execution order, thus WP algorithms themselves are not used in situations where path selection is important or necessary.

Optimization and Implementation. FSE builds VCs in execution order, which allows FSE to easily implement concrete execution of statements not dependent upon symbolic values. The concrete execution can potentially be executed by the underlying hardware (instead of the interpreter) to speed VC generation. For example, in

```
for(i = 0; i < n; i++) ...
```

variables corresponding to `i` in each iteration can be executed concretely and will not be symbolic in the final VC. Other common examples of concrete evaluation in FSE include system calls with concrete arguments, reducing memory operations with concrete addresses to operations on scalar variables, and cases where some input sources are considered symbolic while others concrete (e.g., in a web server we may want to consider reading from the configuration file concretely, but from the network symbolically) [6, 7, 19].

These same optimizations are much more difficult to implement with WP algorithms since they do not process statements in execution order. For example, to achieve the same effects as concrete execution of non-symbolic values, WP algorithms would have to implement constant folding, which typically requires significant additional implementation overhead (e.g., build the CFG, define the data flow analysis, etc.). Other optimizations implemented in FSE (e.g., evaluating system calls when argu-

[1] Note that some implementations do not syntactically create a single formula for all program paths. Instead, they query a theorem prover for each path's VC until a solution is found. Note, however, that querying on each path is equivalent to creating one formula that is the disjunction of all path formulas (see § 6).

ments are concrete) are much more difficult to imagine in WP-based algorithms since they do not build VCs in execution order.

Expressiveness. VCs produced by FSE are quantifier-free, while VCs produced by efficient WP algorithms are universally quantified [15]. Quantifier-free formulas require less sophisticated decision procedures. Thus, all other things being equal, quantifier-free formulas are preferable.

Overall, the implementation advantages of FSE have made it preferable in practice in a tremendous number of application domains despite the worse theoretical VC size. Notable examples include Klee [6]/ EXE [7], Bouncer/Vigilante [10, 11], DART [18], and CUTE [27], all of which choose to use FSE for VC generation. We emphasize that these all choose FSE despite the fact that efficient WP-style VC generation had been previously known [15]. Thus, a natural question is whether we can achieve the benefits of FSE while retaining the smaller VC size of WP-based algorithms.

In this paper, we develop a directionless weakest precondition (DWP) algorithm that enjoys the benefits of both WP and FSE-style VC generation. By directionless, we mean that the algorithm can process program statements in either the forward or backward direction. The final generated VC is the same size as the best WP algorithms: $O(M^2)$ where M is the number of program statements. The directionless nature of the algorithm means that WP-style calculations can run in execution order and thus can take advantage of the above benefits that make FSE attractive in practice while retaining the small VC size. Our algorithm motivates the investigation of FSE/WP correspondences.

Our contributions include:

- A general weakest precondition algorithm (DWP) that can be run in either the forward or backward direction. At a high level, this means that the significant amount of work that currently uses FSE (e.g., [6, 10, 11, 14, 18, 19, 23, 27]) can take advantage of our DWP algorithm to produce smaller VCs. For example, using our approach the FSE VCs at confluence points (i.e. when two branches merge) can be merged, while previous FSE would not merge the formulas. The VCs DWP generates enjoy the same syntactic benefits that make them easier to reason about by automated decision procedures as previous work [15, 21] (§ 3.1).
- We provide formal structural induction-style proofs of correctness and $O(M^2)$ VC size. Previous work argued for $O(M^2)$ VC size, but did not provide complete proofs [15, 21]. Our size bound (§ 3.4) is tighter than previous work [15, 21].
- We prove a correspondence between VCs generated by WP and FSE algorithms. Somewhat surprisingly, the correspondence is not an equivalence: there are some programs for which WP is well-defined (in the common GCL language [15, 21]) but FSE is not. Note this is true for any WP and FSE algorithm over constructs allowed by the guarded command language from [15, 21, 24], not just our algorithm. Thus, it would be incorrect to say (as we initially thought) that FSE is simply a WP calculation with constant folding. The correspondence falls out naturally from the structural induction within our proof, which is an advantage of our proof approach over previous work [15, 21].
- We implement and evaluate our WP algorithm against FSE both in terms of the size of the generated VC, and the time to solve the VC (§ 5). In particular, we evaluate FSE vs. WP within our platform for performing assembly code analysis [3] using two different decision procedures, and within KLEE [6], which is a well-known and public FSE implementation used to generate test cases for programs. Our experiments show that 1) the DWP VCs are significant smaller in practice than FSE, and 2) *DWP VCs can be solved up to* $34\times$ *faster than FSE VCs* for real programs.

$x := e$	Assign variable x to the value of e.
assert e	Aborts if e is not true.
assume e	Only executes when e is true.
$S_1; S_2$	Executes S_1 then S_2.
$S_1 \square S_2$	Executes either S_1 or S_2.

Table 1: Guarded Command Language Statements S.

2. Weakest Preconditions

2.1 The Guarded Command Language

The weakest precondition is calculated over programs written in the guarded command language (GCL). The GCL we use is shown in Table 1, and is identical to that in related work [15, 21]. Although the GCL may look simple, it is general enough that common languages as diverse as Java [15] and x86 assembly [5] can be converted into GCL statements. The translation from typical programming languages into the GCL (and possible trade-offs) is addressed elsewhere (e.g., [15]) and is straightforward. We stress that translating explicitly to the GCL is not needed: the GCL serves as a reference language. Our focus is on creating VCs for programs already in the GCL.

A program in the GCL consists of a sequence of GCL statements. We place no limits on expressions within statements other than they be side-effect free. Typical binary operators, unary operators, memory operations, etc., can all be used. We use T for the expression true, and F for the expression false. A GCL statement can be seen as a mapping between start states and post-execution states. The assignment statement $x := e$ updates the program state so that the location x contains the computed value of e. $S_1; S_2$ denotes a statement sequence. The "choice" statement $S_1 \square S_2$ denotes executing either S_1 or S_2, and corresponds roughly to an "if" command in most languages. assert e does nothing if e is true; otherwise it causes the program to abort. In other words, it maps states where e is true to themselves, and all other states to failure. Finally, assume e statements are partial commands that say the executing program state s has no corresponding output state when e is unsatisfied [24]. This can roughly be viewed as saying that execution will not proceed down a path where an assumewould be unsatisfied. Overall, there are three things that can happen: a program can terminate normally on an input, a program can "go wrong" on an input (failed assert), and a program can be undefined for an input (unsatisfiable assume).

Processing loops. Our approach, like similar previous work (e.g., [6, 15, 17, 21]), targets acyclic GCL programs. If program invariants are available, loops can also be desugared directly into the GCL shown in Table 1 (e.g., using algorithms described in [15]). An arbitrary program can be made acyclic by bounding the number of loop iterations considered. Bounding loop iterations turns the problem of computing such predicates into something that can be done automatically based upon program syntax alone, albeit with results that are with respect to the upper loop bounds.

FSE implementations typically bound the number of loop iterations to guarantee termination and avoid getting "stuck" in potentially long-running or infinite loops. Similarly, we can unroll loops for weakest preconditions as many times as a forward algorithm would execute a loop. Note that bounding loop iterations is common in symbolic execution. For example, suppose n is input in:

```
while((2^n + 3^n) = 7^n) { n++; ... }
```

Forward symbolic execution would build a VC for 1 loop iteration with the branch guard $2^n + 3^n = 7^n$, a VC for the 2nd iteration as $2^{n+1} + 3^{n+1} = 7^{n+1}$, and so on. At each iteration symbolic execution needs to decide whether to execute the loop once again (potentially in an effort to reach an interesting statement inside the loop), or to exit the loop and continue with the rest of the program. Providing upper bounds keeps forward symbolic execution from getting "stuck" in such potentially long running or infinite loops.

An equivalent approach to considering the loop at most N times is to unroll the loop N times (keeping the checks for the loop condition) and eliminate the back edge. We used this for our implementation of DWP since it makes converting to DSA straightforward, and we could reuse the code with other algorithms.

For example, consider the loop while e S. By expanding one loop iteration we get (assume e; S; while e S) $\square$ assume $\neg e$. It can be expanded like this as many times as we like, and then we can replace the loop with an assert so as to not consider more than N iterations.

A common optimization in FSE is to query the theorem prover dynamically to determine whether another loop iteration is possible, to avoid considering the loop more times than necessary. A DWP implementation running in the forward direction and doing DSA conversion dynamically could easily incorporate the same optimization.

2.2 Traditional Weakest Preconditions

The weakest precondition wp for a program P and post-condition Q is a Boolean predicate such that when $wp(P, Q)$ holds, executing P is guaranteed to terminate in a state satisfying Q. By "weakest" we mean that for all other predicates R that result in P terminating in a state satisfying Q, we have $R \Rightarrow wp(P, Q)$.

The weakest precondition is a backward, automatic, syntax driven calculation. As a result, given a GCL program we can *automatically* calculate the weakest precondition. Given a post-condition Q at statement i, we calculate the precondition for executing i such that Q will be satisfied. The calculation is driven by GCL predicate transformers, and there is one for each program statement type. The GCL predicate transformers are shown in Table 2. Most rules are self-explanatory. The rule $Q[e/x]$ indicates substituting all occurrences of x for e in Q.

Example 1. Suppose we have a language where all values and variables are 32-bit unsigned integers, and the following program where x is input:

```
if (x%2 = 1) { s := x+2; } else { s := x+3; }
```

The GCL for this program is:

$$(\text{assume } x\%2 = 1;\ s := x + 2)\square$$
$$(\text{assume } \neg(x\%2 = 1);\ s := x + 3)$$

Let Q be the condition that s will be the result of arithmetic overflow (i.e., $Q = (s < x)$ since s would be greater than x if it didn't overflow and wrap around). The weakest precondition is calculated automatically using the rules from Table 2 as:

$$x\%2 = 1 \Rightarrow ((x + 2) < x) \wedge \neg(x\%2 = 1) \Rightarrow ((x + 3) < x)$$

This predicate is only satisfied by an assignment of values to x that result in overflow. If there is no such assignment (i.e., the predicate is unsatisfiable) then we are guaranteed the program will always terminate in a state such that overflow did not occur.

Of course overflow can happen in the example program when $x \in \{2^{32} - 2, 2^{32} - 1\}$. In practice, satisfiability would be decided by giving the above VC to a decision procedure.

Dijkstra's Differences. Dijkstra's GCL originally did not include assumeand assertstatements, but did include guarded if ... fi, skip, and abortcommands [13]. skipis equivalent to assert T, and abortis equivalent to assert F. Therefore, the omission of Dijkstra's skipand abortare superficial.

Dijkstra's GCL included guarded commands of the form

$$\text{if } e_1 \rightarrow S_1 \square ... \square e_n \rightarrow S_n \text{ fi}$$

which means to select one of the e_i's that are true and execute the corresponding S_i. The if fi command can be desugared into the GCL in Table 1 by using an assumefor each guard for each selected statement:

$$\text{if } e_1 \rightarrow S_1 \square e_2 \rightarrow S_2 \text{ fi} \equiv$$
$$\text{assert } e_1 \vee e_2;\ ((\text{assume } e_1;\ S_1) \square\ (\text{assume } e_n;\ S_n))$$

Note that the assertis needed so that activation when no guard is true leads to abortion [13]. We note that Leino [21] omitted the assert, and came up with a different meaning for wp than Dijkstra [13].

Thus, Table 1 can express every Dijkstra command. However, it turns out that the GCL in Table 1 extends Dijkstra's. In particular, Dijkstra syntactically restricted the GCL to satisfy the "Law of the Excluded Miracle", which states that for every statement S wp(S, F) $\equiv$ F. The Law of the Excluded Miracle corresponds to disallowing partial commands and partial correctness (i.e., the more general use of assume) [24]. An example of such a program is assume F because:

$$\text{wp}(\text{assume F}, \text{F}) \equiv \text{F} \rightarrow \text{F} \equiv \text{T}$$

Our DWP proofs of correctness and size use the general GCL given in Table 1 which admits reasoning about both Dijkstra's GCL programs and more general partial correctness properties. As we will see, partial commands introduce subtle differences such that FSE is only equivalent to the WP for a subset of GCL programs.

2.3 Previous Work in Efficient Weakest Preconditions

The traditional weakest precondition predicate transformers shown in Table 2 can result in a verification condition exponential in the size of the program. The exponential blowup is due to 1) choice statements where the post-condition Q is duplicated along both branches, and 2) substitution in assignment statements. An example of the latter is:

```
wp(x = x+x; x = x+x; x=x+x, x < 10)
```

The final weakest precondition for this program will be $x+x+x+x+x+x+x+x < 10$ due to the substitution rule. The exponential blowup is not just a theoretic nuisance; exponential blowup is commonly encountered in real verification tasks. Duplication in choice statements and assignments have a negative synergistic effect (e.g., if a choice statement $S_1 \square S_2$ preceded the above example all 8 xs would be passed to both S_1 and S_2 branches).

Flanagan and Saxe developed a weakest precondition algorithm that avoids exponential blowup during VC generation [15]. Their algorithm works on assignment-free programs, called passified programs. Their algorithm rests upon the the following equality (as pointed out by Leino [21]):

Theorem 1. For all assignment-free programs P:

$$wp(P, Q) = wp(P, \text{T}) \wedge (wlp(P, \text{F}) \vee Q)$$

where wlp is the *weakest liberal precondition*. The predicate transformers for the weakest liberal precondition are the same as for the weakest precondition except $wlp(\text{assert } e, Q)$, as shown in Table 2. Unlike the semantics of the weakest precondition, the weakest liberal precondition does not require that the program terminate normally; only that if it does terminate, that Q holds. The

S	$wp(S, Q)$	$wlp(S, Q)$	$wp(S, \mathrm{T})$	$wlp(S, \mathrm{F})$
$x := e$	$Q[e/x]$	$Q[e/x]$	T	F
assert e	$e \wedge Q$	$e \implies Q$	e	$\neg e$
assume e	$e \implies Q$	$e \implies Q$	T	$\neg e$
$S_1; S_2$	$wp(S_1, wp(S_2, Q))$	$wlp(S_1, wlp(S_2, Q))$	$wp(S_1, wp(S_2, \mathrm{T}))$	$wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})$
$S_1 \square S_2$	$wp(S_1, Q) \wedge wp(S_2, Q)$	$wlp(S_1, Q) \wedge wlp(S_2, Q)$	$wp(S_1, \mathrm{T}) \wedge wp(S_2, \mathrm{T})$	$wlp(S_1, \mathrm{F}) \wedge wlp(S_2, \mathrm{F})$

Table 2: Weakest precondition and weakest liberal precondition predicate transformers.

key part of Theorem 1 is the post-condition during VC generation is always a constant.

3. Directionless Weakest Preconditions

In this section we first describe an initial directionless weakest precondition algorithm that results in a VC $O(M^4)$ in the number of program statements M. By directionless we mean it can run forward (first to last) or backward (last to first) over a program. We prove correctness, and then show how to extend the algorithm (in a way that maintains correctness) to generate VCs only $O(M^2)$ in size of the original program.

3.1 Initial Directionless Weakest Precondition Algorithm

In this section we focus on producing VCs that (given no additional optimizations) will be syntactically equal to those of Flanagan and Saxe [15] since such VCs have previously been shown more efficient to solve in practice. Unlike previous work such as Flanagan and Saxe, however, our algorithm can take advantage of forward-based optimizations, as described in the next section.

Passified programs. Our algorithm works on passified GCL programs [15, 21]. A passified GCL program is assignment-free. The high-level intuition for removing assignments is to convert them into statements about equality. For simplicity, we assume here as a pre-processing step that we statically passify the program by replacing assignment statements with assume statements. In § 6, we revisit this and discuss performing passifying on-the-fly, such as needed when VC generation is performed dynamically. The algorithm to transform a GCL program into a passified program is:

1. Put P into dynamic single assignment (DSA) form P' where every variable will dynamically be assigned at most once. For acyclic programs, this can be accomplished by using the algorithm in [15], or by converting P to a static single assignment form (SSA) and then removing Φ expressions [21]. The DSA program P' is at most $O(M^2)$ the size of P, where M is the number of statements. The transformation takes $O(M^2)$ time. It is worth noting that the SSA form of a program is, in practice, usually linear in size [1, 12], and ultimately the generated VC will be linear in the size of the DSA program.
2. Replace each assignment statement of the form $x_i := e$ with assume $x_i = e$. The resulting assignment-free program is called a *passified* program.

Then, to calculate the weakest precondition, we can use the fact that if P' is a passified version of P, and $\vec{x}$ are the variables introduced during passification, then $wp(P, Q) = \forall \vec{x} : wp(P', Q)$ as was done in [15]. We later show how to eliminate the universal quantification in § 4.3. [2]

[2] Note universal quantification is necessary for correct semantics, e.g., $wp(x := 5, x < 7) = \forall x.wp(\text{assume } x = 5, x < 7) (\neq \exists x.wp(\text{assume } x = 5, x < 7))$.

F-ASSERT
$$\frac{}{f(\text{assert } e) = ([e], [\neg e])}$$

F-ASSUME
$$\frac{}{f(\text{assume } e) = ([e], [\mathrm{F}])}$$

F-SEQ
$$\frac{f(A) = (n_A, w_A) \qquad f(B) = (n_B, w_B)}{f(A; B) = (n_A @ n_B, w_A @ w_B)}$$

F-CHOICE
$$\frac{\begin{array}{ll} f(A) = (n_A, w_A) & G(n_A, w_A) = (N_A, W_A) \\ f(B) = (n_B, w_B) & G(n_B, w_B) = (N_B, W_B) \end{array}}{f(A \square B) = ([N_A \vee N_B], [W_A \vee W_B])}$$

G-BASE
$$\frac{}{G([n], [w]) = (n, w)}$$

G-REC
$$\frac{G(n_s, w_s) = (N, W)}{G(([n_1] @ n_s), ([w_1] @ w_s)) = (n_1 \wedge N, w_1 \vee (n_1 \wedge W))}$$

DWP
$$\frac{G(f(S)) = (N, W)}{\mathrm{DWP}(S, Q) = \neg W \wedge (N \implies Q)}$$

Figure 1: Our basic algorithm.

Notation. We adopt some conventions from Flanagan and Saxe [15]. N is used for formulas that describe the initial states from which the execution of S may terminate normally (i.e., $N = \neg wlp(S, \mathrm{F})$). W is used for formulas that describe the initial states from which the execution of S may go wrong (i.e., $W = \neg wp(S, \mathrm{T})$). We found it easier to think in terms of N and W than using double negatives and wp/wlp. For example, $S_1; S_2$ can go wrong when (a) S_1 goes wrong, or (b) S_1 terminates normally, but S_2 goes wrong. We write this as $W_1 \vee N_1 \wedge W_2$, which we found more straightforward than $wp(S_1, \mathrm{T}) \wedge wlp(S_1, \mathrm{F}) \vee wp(S_2, \mathrm{T})$. n and w are used for lists of said expressions, respectively.

$[a]$ denotes a list with a as its single element. $a@b$ denotes the concatenation of two lists. (a, b) denotes an ordered pair with a as the first element and b as the second. We use the following meta-syntactic variables: e for expressions, x for variables, Q for a post-condition, and S for GCL statements. We use $\vec{x}$ as the new variables introduced by the DSA transformation.

Since we use the weakest precondition and weakest liberal precondition semantics with respect to the constants true (T) and false (F) extensively in this paper, we show their appropriate values in Table 2.

Basic Algorithm. Figure 1 shows our basic algorithm for calculating the weakest precondition of a passified program P with respect to a post-condition Q, denoted as $\mathrm{DWP}(P, Q)$. We form each statement as an inference rule as this leads to a syntax-based proof of correctness and size. The algorithm can be read as a pattern-match: any time we encounter a statement below the line we perform the calculation on top of the line, returning the corresponding results.

The intuition for DWP is that we can compute sub-formulas for all the individual statements (via f) in either direction, and then later combine them into a VC for the whole sequence (via G). f returns two per-statement VCs: one for successful termination and one for going wrong. The returned VCs are prepended to a list of other VCs generated so far. G takes the two lists of VCs and constructs the appropriate wp or wlp. DWP assembles the final VC for an arbitrary post-condition similar to Theorem 1. Note that *not* calling G on F-SEQ is not a mistake: we instead lazily call G when needed. For example, given $S_1; S_2$ we can compute the formula for S_1 and then S_2, or for S_2 and then S_1. Also note that we are *not* simply renaming the post-condition: no calculation is of the form `let q = Q in wp(S, q)`. As pointed out by Leino, using `let` bindings in this way would only reduce the syntactic size of the formula, but not reduce the burden on the decision procedure [21]. Instead, we ensure that at each step of the computation the post-condition for wp and wlp is always a constant, and that the result is always a let-free expression. This leads to DWP being not only smaller but also faster to solve than wp (see § 5).

Indeed, using DWP one could compute a VC for some first part of the program in the forward direction and then compute a VC for the remainder of the program in the backward direction.

3.2 Proof of correctness

In order for our algorithm in Figure 1 to be correct, we must show it is equivalent to calculating the weakest precondition.

Theorem 2 (Correctness)**.** $\forall S, Q\ \text{DWP}(S, Q) = wp(S, Q)$

In order to prove Theorem 2, we first show that $G(f(S))$ calculates the weakest precondition $\neg wp(S, \text{T})$ as W and $\neg wlp(S, \text{F})$ as N:

Lemma 3. If $G(f(S)) = (N, W)$ then $N = \neg wlp(S, \text{F})$ and $W = \neg wp(S, \text{T})$

Proof. We provide a full proof in our companion TR [20] via structural induction. The case for sequences $S = S_1; S_2$ is the non-trivial step in that it makes the algorithm directionless, and requires some ingenuity to prove. When f is called on S, we do not return the pair N and W. Instead, we return a list of formulas which *later* will be put together into N and W via G. This feature allows us to calculate the formulas for a sequence in any order. We then just call G whenever we need to create the actual VC. The trick in the proof is to first show correctness given a version of F-SEQ that calls G eagerly, and then show that correctness when maintained if called lazily.

□

Proof of Theorem 2. Lemma 3 says that the sub-computations in Figure 1 compute the weakest precondition and weakest liberal precondition. In order to demonstrate correctness we just need to show DWP combines the results correctly. Let $G(f(S)) = (N, W)$. Then

$$\begin{aligned}
&\text{DWP}(S, Q) = \neg W \wedge (N \implies Q) && \text{By rule DWP.}\\
&= \neg\neg wp(S, \text{T}) \wedge (\neg wlp(S, \text{F}) \implies Q) && \text{By Lemma 3.}\\
&= wp(S, \text{T}) \wedge (wlp(S, \text{F}) \vee Q) && \text{Logic}\\
&= wp(S, Q) && \text{By Theorem 1.}
\end{aligned}$$

□

3.3 Efficient Directionless WP

The initial algorithm in Figure 1 may result in a formula quadratic in size of the passified program. Since the passified program is itself $O(M^2)$, the total formula size is $O(M^4)$ (where M is again the size of the original program). In order to see where the quadratic blowup comes from, consider the G-REC rule. G is given a list n of sub-formulas that are required for normal termination, and similarly a list w for wrong termination. The rule computes the pair $N_1 \wedge N$ and $W_1 \vee (N_1 \wedge W)$, thus duplicating N_1. The idea for N_1 in the latter is that a program may go wrong if it goes wrong at statement 1, or if statement 1 terminates but statement 2 goes wrong, and so on. This duplication becomes a real problem when there is a deeply nested series of sequences and choices.

In order to avoid duplicating the expression N_1, we give it a name and refer to the name instead of copying in the actual formula. We do this by tracking some extra state which we refer to as v and V. It is simply a set of (variable, expression) pairs corresponding to expressions that would otherwise be duplicated. For example, if e is an expression that we would be duplicating in the original rules, we instead make up a variable, say x, and use x where we would have used e and return the pair (x, e) along with the resulting formula.

The DWP rule needs to reflect the meaning of the variables introduced:

$$\frac{f(S) = (v, n, w) \qquad G(n, w) = (V, N, W)}{\text{DWP}(S, Q) = (\bigwedge_{(x,e)\in v\cup V} x = e) \wedge \neg W \wedge (N \implies Q)}$$

The only change to individual rules is to G-REC when we introduce $x = e$ to use x instead of duplicating the expression e. The final algorithm is shown in Figure 2.

3.4 Size of the VC

We will prove the size of $\text{DWP}(S, Q)$ is linear in the size of the passified program S by proving an upper bound:

Theorem 4. $|\text{DWP}(S, Q)| < 2 \cdot |S| + 9 \cdot \texttt{len}(S) + |Q|$

Note that previous work argued a weaker bound but did not provide proofs [15, 21]. Since passification may result in quadratic blowup, the final predicate will be in the worse case $O(M^2)$. Recall that our passification process (§ 3.1) does not typically incur the quadratic blowup, and is linear in practice [1, 12]. The full proof is available in the technical report[20]; due to space limitations we could not publish it here. Note that in § 4.4 we show that the VC size can be further reduced for deterministic complete programs.

4. Efficient Forward Symbolic Execution

One of our motivations for developing an efficient, directionless weakest precondition algorithm is to make forward symbolic execution efficient. The main benefit of using our algorithm is the exponentially smaller VCs, shorter VC generation times depending upon the algorithm used for forward symbolic execution, and in our results, shorter verification times. To this end, in this section we lay out:

- The FSE algorithm in § 4.1 as implemented in existing work (e.g. [6, 10, 11, 14, 18, 19, 23, 27]). We also note that FSE is only well-defined on the class of GCL programs we call *deterministic programs*.
- An algorithm for efficient FSE based on DWP in § 4.2.
- A proof of correctness in § 4.3 showing that FSE and DWP produce semantically equivalent VCs. The correctness proof establishes that DWP can be used as a drop-in replacement for FSE.
- We discuss when FSE is not equivalent to WP in § 4.4.

4.1 Semantics of Forward Symbolic Execution

Forward symbolic execution is an algorithm which takes the operational semantics of a language and augments them such that values at each point of execution are in terms of input variables. More precisely, values in the language become expressions. In contrast,

$$\textsc{F-ASSERT}\quad \frac{}{f(\textsf{assert } e) = (\emptyset, [e], [\neg e])}$$

$$\textsc{F-ASSUME}\quad \frac{}{f(\textsf{assume } e) = (\emptyset, [e], [false])}$$

$$\textsc{F-SEQ}\quad \frac{f(A) = (v_A, n_A, w_A) \qquad f(B) = (v_B, n_B, w_B)}{f(A; B) = ((v_A \cup v_B), n_A @ n_B, w_A @ w_B)}$$

$$\textsc{F-CHOICE}\quad \frac{f(A) = (v_A, n_A, w_A) \qquad G(n_A, w_A) = (V_A, N_A, W_A) \qquad f(B) = (v_B, n_B, w_B) \qquad G(n_B, w_B) = (V_B, N_B, W_B)}{f(A \,\square\, B) = ((v_A \cup v_B \cup V_A \cup V_B), [N_A \vee N_B], [W_A \vee W_B])}$$

$$\textsc{G-BASE}\quad \frac{x \text{ is fresh}}{G([N_1], [W_1]) = (\{(x, N_1)\}, x, W_1)}$$

$$\textsc{G-REC}\quad \frac{x \text{ is fresh} \qquad G(n_s, w_s) = (V_s, N_s, W_s)}{G(([N_1]@n_s), ([W_1]@w_s)) = (\{(x, N_1)\} \cup V_s, x \wedge N_s, W_1 \vee (x \wedge W_s))}$$

Figure 2: The final algorithm for efficient, directionless weakest preconditions.

$$\textsc{Var-Sub}\quad \frac{}{x|\sigma \Downarrow_e \sigma(x)}$$

$$\textsc{Fwd-Assert}\quad \frac{e|\sigma \Downarrow_e e'}{\textsf{assert } e|\langle \sigma, \Pi\rangle \Downarrow \langle \sigma, e' \wedge \Pi\rangle}$$

$$\textsc{Fwd-Assign}\quad \frac{e|\sigma \Downarrow_e e' \qquad \sigma' = \sigma[x \mapsto e']}{x := e|\langle \sigma, \Pi\rangle \Downarrow \langle \sigma', \Pi\rangle}$$

$$\textsc{Fwd-Seq}\quad \frac{S_1|\langle \sigma, \Pi\rangle \Downarrow \langle \sigma_1, \Pi_1\rangle \qquad S_2|\langle \sigma_1, \Pi_1\rangle \Downarrow \langle \sigma_2, \Pi_2\rangle}{S_1; S_2|\langle \sigma, \Pi\rangle \Downarrow \langle \sigma_2, \Pi_2\rangle}$$

$$\textsc{Fwd-ite}\quad \frac{e|\sigma \Downarrow_e e' \qquad S_1|\langle \sigma, e' \wedge \Pi\rangle \Downarrow \langle \sigma_1, \Pi_1\rangle \qquad S_2|\langle \sigma, \neg e' \wedge \Pi\rangle \Downarrow \langle \sigma_2, \Pi_2\rangle}{\textsf{if } e \textsf{ then } S_1 \textsf{ else } S_2|\langle \sigma, \Pi\rangle \Downarrow \langle \textsf{merge}(\sigma_1, \sigma_2), \Pi_1 \vee \Pi_2\rangle}$$

Π	Path predicate
σ	Execution context (maps variables to their current symbolic expression in terms of input variables)

Figure 3: A basic forward symbolic execution interpreter built from the operational semantics of the GCL.

concrete execution for the same language would typically execute the program such that values are atomic units (e.g., integers).

Since forward symbolic execution is an execution, it is natural to define the algorithm in terms of the operational semantics of the GCL. We show the operational semantics for a symbolic evaluator in Figure 3.

The semantics of symbolic execution must include the conditions under which a path is executed, called the *path predicate*. More formally, a path predicate is a predicate in terms of input variables which is satisfied by all assignments of concrete values to input variables that would guarantee the path is taken. We use Π to denote the path predicate created so far by symbolic execution. We use σ to denote the current mapping of variables to their symbolic values (i.e., expressions). The rules give the semantics of the evaluation of s with current path predicate Π and state σ, denoted $\langle \sigma, \Pi\rangle$. The result of evaluating (denoted $\Downarrow$) a statement is a new execution context and a new path predicate.

Definition 4.1. Forward symbolic execution of a program P with desired post-condition Q is defined as:

$$\frac{P|\langle \sigma, \mathrm{T}\rangle \Downarrow \langle \sigma', \Pi'\rangle}{\textsc{fwd}(P, Q) : \Pi' \wedge Q}$$

where σ maps variables to their initial symbolic values.

The evaluation rules behave as follows. We denote expression evaluation of e to e' as $e \Downarrow_e e'$. In symbolic execution an expression evaluation merely performs a substitution of used variables with symbolic variables. We show the evaluation substitution rule as VAR-SUB, but otherwise omit expression evaluation rules.

We symbolically evaluate an assignment statement $x := e$ by first evaluating e to a new expression e', and then updating our execution context σ to include the new variable/value binding. A sequence $S_1; S_2$ first evaluates S_1, then evaluates S_2. For example, if $\sigma = \{a \mapsto x_0, b \mapsto x_1\}$ where x_0 and x_1 are symbolic inputs, then $c := a+b; d := 2*c$ evaluates to $\sigma' = \sigma \cup \{c \mapsto x_0+x_1, d \mapsto 2*(x_0+x_1)\}$.

The symbolic evaluation of conditional statements evaluates both branches, shown in the FWD-ITE rule. In contrast, during concrete execution the branch guard e is evaluated and then the single appropriate branch is selected for subsequent execution. We want the ability to execute both branches so that the formula generated captures any possible path through the program. In order to do this, we first evaluate the branch guard e to a symbolic expression e'. e' will be purely in terms of (symbolic) input variables. We then assert that e must be true along the true branch S_1 by adding it to the path predicate. The result is $e \wedge \Pi$ means that to execute statement s, everything to reach the condition in Π must be true, and finally e must be true. We similarly execute the false branch by adding $\neg e$ to the path predicate.

Deterministic and Partial Programs. As alluded to earlier, unrestricted use of assumecan lead to partial programs [24] as well as nondeterministic behavior when combined with $\square$ [13]. We say a GCL statement is *partial* if it does not map all possible pre-execution states to a post-execution state. Conversely, we say it is *complete* if it does map all pre-execution states to at least one post-execution state.

We say a program is *deterministic* if every choice statement in it is of the form assume e; $S_1 \,\square$ assume $\neg e$; S_2. We say a program is *nondeterministic* if it has some choice where the guards are not mutually exclusive (i.e. when there exist states from which both branches are allowed).

Our definition of FSE is restricted to complete, deterministic programs as the semantics of FSE on this class of program are generally agreed upon. In this section, we use if e then S_1 else S_2 as syntactic sugar for assume e; $S_1 \,\square$ assume $\neg e$; S_2 since all complete, deterministic programs can be expressed in this form. Although particular implementations of forward symbolic execution in related work may be nuanced, the restricted language still reflects the core constructs in all FSE applications we are aware of (e.g., [2, 4, 6, 7, 10, 11, 17–19, 23, 28]).

We revisit adding assumestatements in § 4.4 after we prove DWP(S) = FWD(S) for complete, deterministic programs S, as it is particularly informative to see why the proof of equivalence does not work between WP and FSE algorithms when assumeis unrestricted.

Problem 1: Merging Paths One issue with forward symbolic execution is how best to handle the confluence point after executing

a conditional branch. Consider a program of the form:

$$S_1; \text{if } e \text{ then } S_2 \text{ else } S_3; S_4$$

Suppose we are interested in generating a formula to reach S_4. At the branch point either S_2 or S_3 could be taken. Since either path could be taken to reach subsequent statements, it is natural to have the path predicate be the disjunction of paths at confluence points such as S_4. The FWD-ITE rule reflects this: if Π_1 is the path predicate after executing one side of a branch, and Π_2 is the predicate for the other side, then the path predicate at the confluence point is $\Pi_1 \vee \Pi_2$.

Deciding how to merge the execution contexts σ_1 and σ_2 at the confluence of two execution paths, however, is not so straightforward. We denote the *problem* of merged paths by the merge function. For example, suppose along one path we have $\sigma_1 = \{a \mapsto 2 * x_0\}$ and along the other path we have $\sigma_2 = \{a \mapsto 3 * x_0\}$. What value should a have at the confluence of these two paths?

In concrete execution only one of the two paths at branches will be taken; thus there is no problem with merging states. The most straightforward solution is to mimic concrete execution during symbolic execution and "fork off" two different engines, one for each path. The final predicate will still be the disjunction of each path predicate. A forking algorithm, however, will lead to an exponentially large predicate and an exponential run time. For example, consider:

$$\text{if } e_1 \text{ then } S_1 \text{ else } S_2; \text{if } e_2 \text{ then } S_3 \text{ else } S_4; \text{if } e_3 \text{ then } S_5 \text{ else } S_6; S_7$$

In this situation there are 8 program paths, and we will need to execute each one individually in order to create a path predicate for S_7. The final predicate will be the disjunction of all 8 paths. In the worst case, every branch doubles the run time, and doubles the size of the final VC. Despite the exponential behavior, forking off the interpreter is a common solution (e.g., [6, 7, 10, 16, 18, 22, 28] and many others).

Problem 2. Formula Size is Exponential. A second issue with forward symbolic execution is the resulting formula may be exponential in size. One reason is the substitution performed by VAR-SUB, which is analogous to the substitution performed by the weakest precondition. Just as before, the program $x := x + x; x := x + x; x := x + x$ will produce a final formula that contains x 8 times. Also, the final predicate size doubles every time we encounter a conditional branch in FWD-ITE since a copy of Π is passed along both branches.

Advantage: Mixed Execution. Given the significant theoretical performance issues with forward symbolic execution, why would it ever be used in practice?

One significant advantage is that it is relatively easy to modify a symbolic execution engine to perform a combination of symbolic and concrete execution. For example, we could alter our operational semantics such that we keep both a context for variables with concrete values σ^c and a symbolic context for variables with values in terms of symbolic inputs σ^s. When evaluating an expression, instead of merely replacing all occurrences of variables with their current symbolic formula, we first try to evaluate them concretely.

Since mixed execution takes advantage of initial constants in the program, one may wonder why getting rid of such constants is easier using forward symbolic execution. The answer is that we encounter the constant variable definitions before uses in the forward direction, but not in the backward direction. Furthermore in the forward direction we can simply execute the program concretely on a native processor and only do the symbolic execution in software. Recently several research projects in automated test case generation have implemented mixed execution in this manner and as a result are achieving significant code coverage, e.g., 90% on average in the Linux core-utils package [6].

One may wonder why such constants appear in programs, e.g., why were constants not removed during compilation? While the answer depends upon implementation and application domain, the "constants" are often simply just inputs we do not care about executing symbolically. For example, suppose we wish to generate test cases for the following program:

```
1.  environment = read_configuration(file);
2.  request = get_request(network);
3.  handle_request(environment, request);
```

How we handle a request is parameterized by both the URL and the web-server environment (e.g., a request for a URL "foo" may be a valid request under one configuration, but an invalid request under another).

In a particular application domain, however, we may choose to fix `environment` or `request` to have a particular concrete value. For example, suppose we want to generate test cases for `handle_request`. If we fix `environment` to have a specific concrete value (e.g., the value on the machine that the test case generation is running on), the formulas generated by symbolic execution of `handle_request` will have fewer variables; thus likely be easier to solve for a decision procedure. This in turn would probably result in higher coverage for statements inside `handle_request` than if we left `environment` symbolic.

Advantage: No Universal Quantifiers A second advantage of forward symbolic execution is that the generated predicates do not contain universal quantifiers, while efficient weakest preconditions do. Since we typically give generated formulas to a theorem prover, we want formulas to be amenable to as many theorem provers as possible. By removing universal quantifiers we open up VC verification to a wider variety of theorem provers.

4.2 Efficient Forward Symbolic Execution

In this section we describe our algorithm for using our directionless weakest precondition as a drop-in replacement for forward symbolic execution.

Let P be a program using only the language constructs from Figure 3. Our algorithm for efficiently performing forward symbolic execution for a post-condition Q is:

Step 1. Put P in DSA form. Although the straightforward implementation of this is static (§ 3), we discuss how it can be done dynamically in § 6. Let P' be the DSA program.

Step 2. Calculate DWP(P',Q).

Forward Execution of DSA Program P'. In order to prove that DWP(P',Q) is correct, we first need to define what forward symbolic execution would produce. We show correctness in two steps. First, we describe a simpler form of forward symbolic execution for DSA programs. The second step is to show that forward symbolic execution on DSA programs produces a formula that is logically equivalent to our directionless weakest precondition.

We note that the forward symbolic execution rules in Figure 3 perform substitution because each variable name does not reflect a unique definition. For example, in $x := x + x; x = x + x$ we want any post-conditions to refer to the second x, not the first. In DSA form, however, each variable is assigned only once per path. Thus, instead of performing substitution, each variable can be referred to by name when needed. In the above example, the DSA form is $x_1 = x_0 + x_0; x_2 = x_1 + x_1$, and the post condition can refer to the final value as x_2.

Thus, forward symbolic execution need not perform substitution on programs in DSA form. Further, we need not keep track of the current symbolic state: we can simply refer to the particular variable definition we need. As a result, forward symbolic execution of

$$s(S_1; S_2) = s(S_1) \wedge s(S_2) \qquad s(\textsf{assert } e) = e$$
$$s(\textsf{if } e \textsf{ then } S_1 \textsf{ else } S_2) = (e \wedge s(S_1)) \vee (\neg e \wedge s(S_2))$$
$$s(x := e) = (x = e) \qquad \textsc{fwd}'(P', Q) = s(P') \wedge Q$$

Figure 4: Forward symbolic execution of a program in DSA form.

a DSA program eliminates the problems associated with merge. We show the simplified rules in Figure 4.

Let $\textsc{fwd}(P, Q)$ be forward symbolic execution of P such that an interpreter is forked for each execution path, i.e.:

$$\textsc{fwd}'(P, Q) = \bigvee_{\forall \text{ paths } \pi \in P} \textsc{fwd}(\pi, Q) \tag{1}$$

Then $\textsc{fwd}(P, Q)$ is the same as $\textsc{fwd}'(P', Q)$ using the rules in Figure 4. The correspondence can be shown by simple inspection of each type of program statement. In particular, note that instead of doing substitution in assignment, we simply add a variable-value definition to our logical formula that reflects the state update.

4.3 Correctness

Correctness states that whatever VC forward symbolic execution would calculate is logically equivalent to what the predicate transformers for weakest preconditions would calculate.

Theorem 5. For all complete, deterministic programs P and all predicates Q: $wp(P, Q) = \textsc{fwd}'(P, Q)$

This equivalence means our improvement to weakest preconditions is also an improvement to forward symbolic execution. In particular, since $\textsc{dwp}(P, Q) = wp(P, Q)$, our directionless algorithm can therefore be used as a drop-in replacement for FWD.

In order to prove this, we need to establish two things. First, recall that weakest preconditions are universally quantified, while forward symbolic execution is only existentially quantified. Thus, we will need to establish a correspondence between the two:

Lemma 6. $\forall x : (x = e \Rightarrow Q) \iff \exists x : (x = e \wedge Q)$

Proof. Full proof omitted. (This equivalence is easy to verify, eg. via automated theorem proving.) □

Note that weakest precondition is defined as:

$$\begin{aligned} wp(P, Q) &= wp(P, \mathrm{T}) \wedge (wlp(P, \mathrm{F}) \vee Q) \\ &= (wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F})) \vee (wp(P, \mathrm{T}) \wedge Q) \end{aligned}$$

The intuition behind our proof of Theorem 5 is that the first part of this disjunction is always false for complete, deterministic programs, and the second part of the conjunction corresponds to forward symbolic execution. We prove the first part of this in the following lemma:

Lemma 7. For all complete, deterministic programs P:

$$wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) = \mathrm{F}$$

Proof. By induction on the structure of P:

Case: $P = \textsf{assert } e$

$$\begin{aligned} & wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) = e \wedge \mathrm{T} \wedge (e \Rightarrow \mathrm{F}) \\ & = e \wedge \neg e = \mathrm{F} \end{aligned}$$

Case: $P = x := e$ [3]

$$\begin{aligned} & wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) \\ & = wp(P, \mathrm{T}) \wedge \forall x, (x = e \Rightarrow \mathrm{F}) \\ & \implies wp(P, \mathrm{T}) \wedge (e = e \Rightarrow \mathrm{F}) && (\forall \text{ elimination with } e \text{ for } x) \\ & = \mathrm{F} \end{aligned}$$

Case: $P = \textsf{if } e \textsf{ then } S_1 \textsf{ else } S_2$ [4]

$$\begin{aligned} & wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) \\ & = (e \Rightarrow wp(S_1, \mathrm{T}) \wedge \neg e \Rightarrow wp(S_2, \mathrm{T})) \\ & \quad \wedge (e \Rightarrow wlp(S_1, \mathrm{F}) \wedge \neg e \Rightarrow wlp(S_2, \mathrm{F})) \\ & = e \Rightarrow (wp(S_1, \mathrm{T}) \wedge wlp(S_1, \mathrm{F})) \\ & \quad \wedge \neg e \Rightarrow (wp(S_2, \mathrm{T}) \wedge wlp(S_2, \mathrm{F}) \\ & = e \Rightarrow \mathrm{F} \wedge \neg e \Rightarrow \mathrm{F} && \text{By IH.} \\ & = \mathrm{F} \end{aligned}$$

Case: $P = S_1; S_2$

$$\begin{aligned} & wp(P, \mathrm{T}) \wedge wlp(P, \mathrm{F}) \\ & = wp(S_1, \mathrm{T}) \wedge (wlp(S_1, \mathrm{F}) \vee wp(S_2, \mathrm{F})) \\ & \quad \wedge (wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})) \\ & = wp(S_1, \mathrm{T}) \wedge wlp(S_1, \mathrm{F}) \vee wp(S_1, \mathrm{T}) \\ & \quad \wedge wp(S_2, \mathrm{F})) \wedge (wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})) \\ & = \mathrm{F} \vee wp(S_1, \mathrm{T}) \wedge wp(S_2, \mathrm{F}) \\ & \quad \wedge (wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})) && \text{By IH.} \\ & = wp(S_1, \mathrm{T}) \wedge wp(S_2, \mathrm{F}) \wedge wlp(S_1, \mathrm{F}) \\ & \quad \vee wp(S_1, \mathrm{T}) \wedge wp(S_2, \mathrm{F}) \wedge wlp(S_2, \mathrm{F}) && \text{By IH.} \\ & = \mathrm{F} \vee \mathrm{F} && \text{By IH.} \\ & = \mathrm{F} \end{aligned}$$

□

We now prove Theorem 5.

Proof of Theorem 5. By induction on the structure of P, using s from Figure 4:

Case: $P = \textsf{assert } e$

$$e \wedge Q = e \wedge Q \;\therefore wp(P, Q) = s(P) \wedge Q$$

Case: $P = x := e$

$$\begin{aligned} & (1)\ wp(P, Q) = \forall x, (x = e \implies Q) \\ & (2)\ s(P) \wedge Q = (x = e) \wedge Q \\ & \quad = \exists x, (x = e \wedge Q) \\ & (1) = (2) && \text{By Lemma 6} \end{aligned}$$

Case: $P = \textsf{if } e \textsf{ then } S_1 \textsf{ else } S_2$

$$\begin{aligned} & wp(P, Q) \\ & = (e \Rightarrow wp(S_1, Q)) \wedge (\neg e \Rightarrow wp(S_2, Q)) \\ & = e \wedge wp(S_1, Q) \vee \neg e \wedge wp(S_2, Q) \\ & = e \wedge s(S_1) \wedge Q \vee \neg e \wedge s(S_2) \wedge Q && \text{By IH.} \\ & = s(P) \wedge Q \end{aligned}$$

Case: $P = S_1; S_2$

$$\begin{aligned} & wp(P, Q) \\ & = wp(S_1, T) \wedge (wlp(S_1, F) \vee wp(S_2, Q)) \\ & = wp(S_1, T) \wedge (wlp(S_1, F) \vee wp(S_2, \mathrm{T}) \wedge (wlp(S_2, \mathrm{F}) \vee Q)) \\ & = wp(S_1, T) \wedge wlp(S_1, F) \\ & \quad \vee wp(S_1, T) \wedge wp(S_2, \mathrm{T}) \wedge (wlp(S_2, \mathrm{F}) \vee Q)) \\ & = wp(S_1, T) \wedge wp(S_2, \mathrm{T}) \wedge (wlp(S_2, \mathrm{F}) \vee Q)) && \text{By Lemma 7} \\ & = wp(S_1, T) \wedge wp(S_2, \mathrm{T}) \wedge Q && \text{By Lemma 7} \\ & = s(S_1) \wedge s(S_2,) \wedge Q && \text{By IH.} \\ & = s(P) \wedge Q \end{aligned}$$

[3] Since the program is in DSA form, we are simply treating assignments as assumes.

[4] Note that applying the previous definition of WP to the desugared statements we get that

$$wp(\textsf{if } e \textsf{ then } S_1 \textsf{ else } S_2) = (e \Rightarrow wp(S_1, Q)) \wedge (\neg e \Rightarrow wp(S_2, Q))$$

□

4.4 FSE vs WP

Theorem 5 only shows that FSE is the same as WP for deterministic, complete programs. In order to understand the difference between WP and FSE, suppose we tried to prove the FSE/WP correspondence theorem for all programs allowed by the GCL in Table 1, not just deterministic programs as we have shown. As such, we would have to include assumestatements into the proof of Lemma 7 (i.e., show that $wp(\mathsf{assume}\ e, \mathrm{T}) \wedge wlp(\mathsf{assume}\ \neg e, \mathrm{F}) = \mathrm{F}$). However, this statement is clearly false for partial programs, e.g.:

$$\begin{aligned} & wp(\mathsf{assume}\ x = e, \mathrm{T}) \wedge wlp(\mathsf{assume}\ x = e, \mathrm{F}) \\ & = \mathrm{T} \wedge \neg(x = e) \\ & = \neg(x = e) \end{aligned}$$

Since $\neg(x = e) \neq \mathrm{F}$, the Lemma falls apart, and the equivalence theorem does not work. Similarly, the proof for Theorem 5 breaks down for nondeterministic programs, since the obvious extension to FSE would cause it to generate formulas that are satisfiable for states that *could possibly* satisfy the postcondition, whereas WP guarantees the post condition would be satisfied.

In fact, Theorem 5 establishes the correspondence $wp(P, Q) = wp(P, \mathrm{T}) \wedge Q = FSE(P, Q)$ for deterministic programs. Therefore, if we choose to drop in DWP for FSE, we need not compute the wlp (N in [15] and our algorithm) at all, and the generated VC will be $|P| + |Q|$. Note: this means that for deterministic programs, we can calculate an equivalent VC that is only half the size of [15, 21] and Theorem 4.

Overall, the WP and FSE generate equivalent formulas for deterministic programs, including GCL programs using Dijkstra's original semantics. Adding assumestatements, as done by [15, 21], makes them different because assumeintroduces partial relationships while FSE VCs define a total relationship.

5. Evaluation

Setup. We have implemented our DWP in the BAP infrastructure for program analysis on low-level binary code [3]. Our tests with BAP were performed on disassembled compiled binaries for the GNU coreutils programs, limiting ourselves to leaf functions in order to keep the implementation simple (e.g., not having to handle recursive procedures). BAP lifts the binary code to an intermediate language that is similar to a compiler IR. Our FSE implementation worked directly on the BAP IL, whereas for DWP, WP, and Flanagan & Saxe's algorithm (denoted FS) [15] we translated the BAP IL to the GCL as part of generating the formula. We excluded boilerplate functions such as _fini, __i686.get_pc_thunk.bx, __umoddi3, etc., as well as functions both DWP and FSE could solve trivially. We call a case *trivial* where generation + solving time were less than 0.01s, as that meant at least one of the measured times was 0. We omitted these cases because the low time measurement makes the error bounds greater than the measured values, and they are not very interesting (since they are trivial). Since many functions are duplicated in many of the coreutils programs, we considered only a single instance of functions with the same name.

We tested each function with 3 different post-conditions and 2 decision procedures: CVC3 and Yices. We used multiple decision procedures because the time to solve can be vary significantly depending upon the heuristics employed by the decision procedure algorithms. We compared VCs with each post-condition and each decision procedure generated with DWP, FSE, FS, and the WP algorithm as defined in Table 2.

The first post-condition was simply true, i.e., is there an input that will cause this program to terminate? The second post-condition asserted that overflow never occurred in the function (i.e., the x86 OF flag was never set). The third post-condition, denoted $RA \neq RA_0$, checked that the return address was overwritten, eg., if there was a buffer overflow. In both FSE and WP we bounded loops to at most 32 iterations. One might expect FSE to be the best with post-condition true since it simply needs to find a valid path. The no-overflow post-condition is a little harder than the first case since paths that require an overflow are no longer acceptable. The third case is the only one with a non-trivial post condition, so we expect it would lead to more complicated (and more realistic) VCs with all approaches. However, in our experiments DWP outperforms FSE in both VC size and total time to solve in all cases.

When generating VCs for each function we limited each algorithm to 5 minutes of CPU time, 1GB of memory, and 1GB file size for the formula. We also gave the solver another 5 minutes and 1GB of memory for solving formulas. These limits were put in place because earlier versions of the tests would run for many hours before filling up the disk and failing, and most of the time was spent in small subset of the functions. `sha1sum` and `md5sum` in particular had many functions that would cause the process to take a long time, usually thrashing. All measurements were performed on a 2.4 GHz Core 2 processor with 4 GB of RAM. Timing was done with the `time` command on a Linux system, which measures the execution time down to the hundredth of a second and counts the entire time the program runs (from load to exit), including parsing, writing, and GC overhead. Our test suite will be made available at `http://security.ece.cmu.edu`.

Q=true In the first set of experiments, for post-condition true, we had 49 trivial cases, leaving us with 63 nontrivial ones. DWP, FSE, and FS all generated 36 VCs, but WP only 33. Total sizes for the formulas were 1.2MB for DWP, 290MB for FSE, 1.9MB for FS, and 5.6MB for WP. All formulas except one could be solved with either theorem prover. CVC3 timed out while trying to solve the VC generated by FS for search_cur_bkref_entry() from `tac`, but Yices solved it in 0.1s.

On average DWP was 13.2 times faster than FSE using Yices. With CVC3 DWP was only 2 times faster, because CVC3 was much slower to solve both formulas. As is shown in Figure 5, DWP was faster in all but 4 cases.

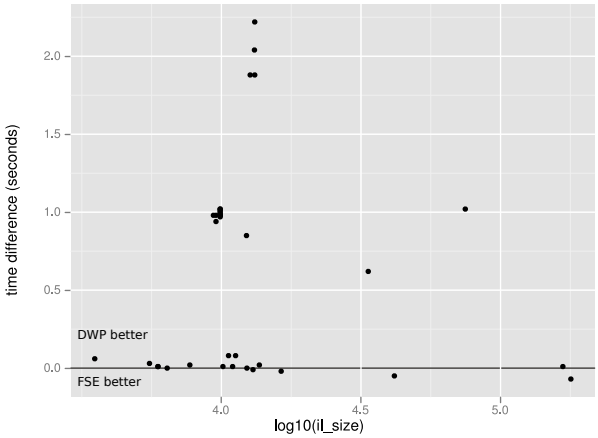

Figure 5: Difference in time to generate and solve with DWP or FSE for $Q = true$. (Positive values mean DWP was faster, negative mean FSE was faster.)

No overflow. For the second set of experiments, asserting no overflow, we had 47 trivially solved functions leaving 65 to consider.

	Mean	Median	Max	SD	Sum	#
DWP gen	0.04	0.02	0.17	0.04	1.33	36
FSE gen	0.66	0.83	2.24	0.66	23.79	36
WP gen	0.05	0.02	0.61	0.11	1.59	33
FS gen	0.05	0.03	0.21	0.05	1.64	36
DWP CVC	1.56	1.48	19.41	3.13	56.09	36
FSE CVC	2.83	1.24	5.98	2.71	101.78	36
WP CVC	1.77	0.75	3.50	1.65	58.42	33
FS CVC	3.57	3.08	12.20	3.68	124.95	35
DWP Yices	0.02	0.02	0.08	0.02	0.56	36
FSE Yices	0.02	0.00	0.04	0.02	0.65	36
WP Yices	0.02	0.03	0.05	0.02	0.66	33
FS Yices	0.05	0.06	0.10	0.03	1.71	36
DWP+CVC	1.60	1.50	19.58	3.15	57.42	36
FSE+CVC	3.49	1.92	8.22	3.29	125.57	36
WP+CVC	1.82	1.36	3.54	1.64	60.01	33
FS+CVC	3.61	3.25	12.25	3.68	126.42	35
DWP+Yices	**0.05**	0.04	0.22	0.05	**1.89**	36
FSE+Yices	**0.68**	0.84	2.27	0.68	**24.44**	36
WP+Yices	0.07	0.05	0.61	0.11	2.25	33
FS+Yices	0.09	0.10	0.27	0.06	3.35	36

Table 3: Times for Q=true

DWP generated 38 VCs (totaling 2.1MB), FSE 36 (981MB), FS 38 (3.0MB), and WP 35 (5.7MB). With CVC3, DWP solved 37, FSE 34, FS 16, and WP 33. Yices again solved all the generated formulas. DWP worked on 2 cases where FSE didn't (sha1_read_ctx from `sha1sum` and set_char_quoting from `yes`).

With Yices DWP was 34 times faster this time, but not quite twice as fast with CVC3. Figure 6 shows that the time differences are greater which is expected since the total solve times are also greater. For every function, DWP was either clearly faster or at least not much slower. In practice, the run time of WP would be either significantly longer than DWP (by not timing out on cases where it currently fails), or WP would end up verifying fewer VCs than DWP.

RA $\neq$ ***RA***$_0$ For the third set of experiments, checking for different return addresses, there were no trivial cases. DWP generated 85 VCs (totaling 2.7MB), FSE 79 (1.1GB), FS 85 (3.9MB), and WP 82 (6.3MB). With CVC3, DWP solved 83, FSE 77, FS 76, and WP 80. Yices was again able to solve all generated VCs. DWP again solved everything FSE solved. FSE couldn't generate formulas for `md5sum` md5_read_ctx, `sha1sum` sha1_read_ctx, `tac` re_string_construct_common, `tail` record_open_fd, `vdir` hash_reset_tuning, and `yes` set_char_quoting, in all but the last due to running out of memory.

In this case DWP was 3.2 times faster than FSE with Yices, but not as nice with CVC3. Both did better with Yices than with CVC3 so it makes more sense to use Yices. The comparison is slightly better for DWP if we compare only the times for VCs both could generate. Figure 7 shows there are a few more cases where FSE did a tiny bit better, but DWP still does better on most of them, sometimes by a lot.

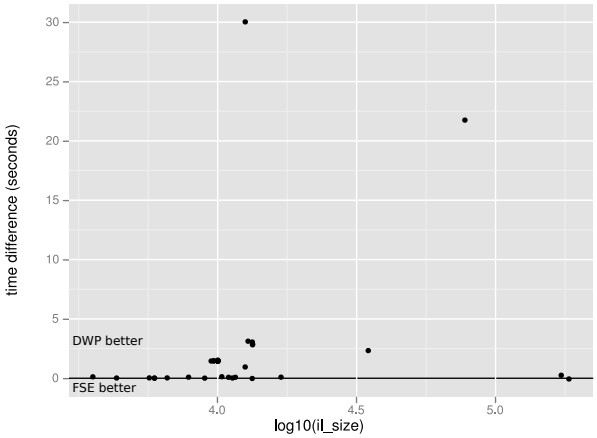

Figure 6: Difference in time to generate and solve with DWP or FSE for No overflow.

	Mean	Median	Max	SD	Sum	#
DWP gen	0.03	0.02	0.19	0.04	2.22	85
FSE gen	1.08	0.05	33.63	4.23	85.61	79
DWP CVC	10.99	1.20	234.06	30.39	923.33	84
FSE CVC	0.94	0.18	22.90	2.79	73.33	78
DWP Yices	0.33	0.16	3.32	0.57	28.27	85
FSE Yices	0.08	0.00	1.50	0.25	6.16	79
DWP+CVC	11.02	1.21	234.23	30.41	925.52	84
FSE+CVC	1.61	0.22	38.33	4.83	125.31	78
DWP+Yices	**0.36**	0.18	3.35	0.58	30.49	**85**
FSE+Yices	**1.16**	0.08	34.01	4.34	91.77	**79**

Table 4: Times for $RA \neq RA_0$.

Klee We have also a preliminary implementation of DWP in Klee [6], an open-source and competitive FSE implementation used to generate test cases for C programs. In our experiments, as the number of branches increased, the more DWP outperformed the native FSE implementation. For example, a typical problem in Klee is to check calls to `strcpy` for out-of-bounds buffer writes given a symbolic source buffer of fixed length n:

```
void strcpy(src, dst) {int i =0;
while(src[i] != NULL){ dst[i] = src[i]; i++;} ...}
```

Klee implements the check by generating a path formula for all possible n iterations. Figure 8 shows how DWP improves performance as n increases. Overall, the time difference between Klee's default FSE and our DWP implementation roughly corresponds to the quadratic vs. exponential difference in the generated VC size. Due to space, we refer interested readers to our companion report for more details and experiments [9].

Summary In our experiments DWP was 2× to 34× faster than FSE, depending on the decision procedure and post-condition. Since CVC3 is slower regardless of whether you use DWP or FSE, it makes more sense to use Yices, in which case the difference between DWP and FSE is even greater. DWP also solved the most

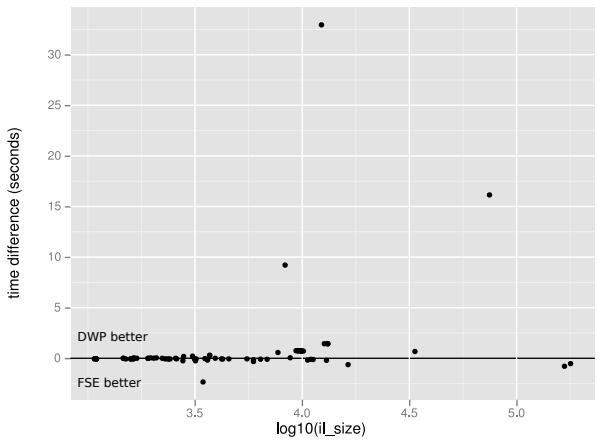

Figure 7: Difference in time to generate and solve with DWP or FSE for $RA \neq RA_0$. (Positive values mean DWP was faster, negative mean FSE was faster.)

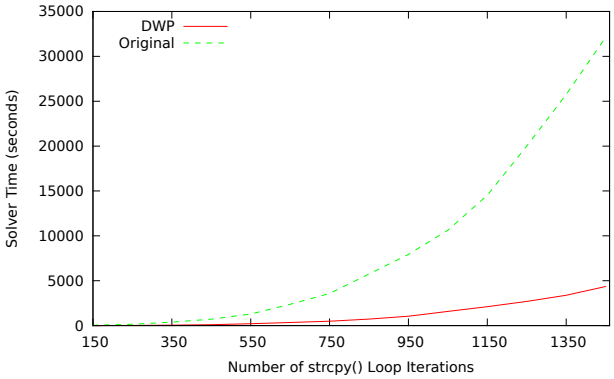

Figure 8: KLEE solver times with DWP and without.

VCs in all cases and against other VC generation algorithms, which in practice is a critical factor.

6. Discussion

Dynamic single assignment. Forward symbolic execution can be implemented in a way that VCs are generated at run-time (e.g., as in KLEE [6], DART [18], and CUTE [27]). Our algorithm acts as a drop-in replacement, but only when the program is first put in a dynamic single assignment form. Our assumed method for putting the program in DSA form via an SSA-like transformation, however, is static. We expect in most application domains this small amount of static analysis is acceptable, e.g., EXE is implemented on top of a compiler which already supports such transformers.

It is possible, however, to use our DWP in a purely dynamic setting by calculating the DSA form of the program on the fly. On-the-fly DSA form requires we add an additional current incarnation context to the execution engine that maps a variable name to its current DSA incarnation. Sequential statements are trivial; every assignment simply uses the context when processing expressions and updates the context on assignment. Branches require that we merge contexts at the confluence point. For example, if in branch 1 the current incarnation of x is x_4, and along branch 2 the current incarnation is x_5, we simply need to add an assignment that $x_6 = x_4$ along the first branch and $x_6 = x_5$ along the second, and then use x_6 as the canonical identifier for x thereafter.

Equivalence between forward symbolic execution and weakest precondition. At a high level it may be unsurprising that weakest preconditions and forward symbolic execution are related. After all, both are calculating a formula that should be satisfied by inputs that cause termination in the desired state. However, we are unaware of any previous work that spells out the differences.

Our algorithm and proof show that weakest precondition is equivalent to forward symbolic execution for typical structural programs. The proof precisely identifies the difference: they are equivalent if the program 1) acts deterministically and 2) does not use assumptions for anything other than branch guards. It may be possible to extend FSE to work on partial programs or nondeterministic ones, but we are not aware of any previous work that extends it in a way that is equivalent to WP.

Eagerly calling theorem provers. Some applications of forward symbolic execution eagerly call a theorem prover when the VC for each program path is first generated. For example, an application such as automated test case generation may only care about a single path to the desired program point and hope that eagerly calling the theorem prover will result in a satisfiable answer before all paths are enumerated. Of course in the worse case all paths must be enumerated before finding a satisfiable one. In such a circumstance the application would have done better by using the DWP directly.

However, the DWP can be advantageous for combining path formulas even when an application typically calls the theorem prover eagerly. For example, in if e then S_1 else $S_2; S$ it would be naive to explore the rest of the program S when $e; S_1$ is initially taken, and then again consider the rest of the program S when $\neg e; S_2$ is taken. Instead, they benefit from creating a single compact VC after the if-then-else that considers all paths, which is then used when subsequently exploring S. Our DWP can be used as a replacement for previous work that tries post-hoc to create compact VCs for two different path formulas [10, 11].

Rearranging formulas During our experiments, we found that some trivial changes to formulas would make them much easier or harder to solve. For example, if we were to use $A \wedge B$ rather than $B \wedge A$, where A was the formula for part of the program that would be executed earlier and B for the part that would be executed later, in some cases one would perform much better (10x faster) than the other, and in other cases the other would. This was unexpected, since in both cases the theorem prover would need to prove both A and B. In our experiments we used default generation order.

7. Related Work

Flanagan *et al.* [15] created the first algorithm for calculating the weakest precondition that is $O(M^2)$ in size. We use the same desugaring and approach to passification as proposed in that paper. They first note that by structural induction using something similar to Theorem 1 they can get a formula $O(M^4)$ in size, and then by renaming duplications in the wlp reduce the size to $O(M^2)$ (although the quadratic bound is argued, it is never formally proven). The actual statement of Theorem 1 was first proposed and proven by Leino [21]. We follow Brumley *et al.* [5] in forming our algorithm as a deductive system. Our motivating is to make proofs syntax-based where the induction hypotheses corresponds to the premise/sub-computation. We also use the fact that $wlp(S_1; S_2, \mathrm{F}) = wlp(S_1, \mathrm{F}) \vee wlp(S_2, \mathrm{F})$ as shown [5]. Snugglebug [8] and Bouncer [10] perform post-hoc simplification when combining single-path formulas.

A recent survey of symbolic execution techniques and applications is provided in [25]. Our formalization is motivated by recent work in automated test case generation such as [6, 7, 17–19], although it is not specific to only those applications. While we restrict ourselves to symbolic execution where loops are bounded, our techniques also work when invariants are available by desugaring loops with invariants as in [15]. Special cases such as when loops and any loop-dependent side-effects can be written as a system of linear equations (e.g. [26]) are related and can possibly handled by a similar approach, but are not directly in the scope of this work.

8. Conclusion

We have shown the first efficient directionless weakest precondition algorithm where the order program statements are evaluated by the algorithm does not matter. We generate verification conditions at most $O(M^2)$ where M is the size of the program. In particular, our algorithm can take advantage of optimizations found in forward symbolic execution, which was previously not possible using weakest preconditions. We show equivalence between our algorithm, previous weakest precondition approaches that could only process statements last-to-first, and forward symbolic execution. The equivalence highlights a deep connection between forward symbolic execution and weakest preconditions where improvements in one algorithm will benefit the other. One implication is that we prove our algorithm can used as a drop-in replacement for forward symbolic execution engines with the benefit that VCs are exponentially smaller than existing forward symbolic execution approaches.

References

[1] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] T. Ball. Abstraction-guided test generation: A case study. Technical Report MSR-TR-2003-86, Microsoft Research, 2003.

[3] Binary Analysis Platform (BAP). http://bap.ece.cmu.edu.

[4] D. Beyer, A. Chlipala, R. Majumdar, T. Henzinger, and R. Jhala. Generating tests from counterexamples. In *Proceedings of the ACM Conference on Software Engineering*, 2004.

[5] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2007.

[6] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2008.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*, Oct. 2006.

[8] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. *SIGPLAN Not.*, 44(6):363–374, 2009.

[9] J. Cooke. Directionless weakest preconditions in klee. Master's thesis, Carnegie Mellon University, Information Networking Institute, 2010. Thesis Advisor: David Brumley.

[10] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of the ACM Symposium on Operating System Principles*, Oct. 2007.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating System Principles*, 2005.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Symposium on Principles of Programming Languages*, 1989.

[13] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[14] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis*, pages 151–162, 2007.

[15] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the Symposium on Principles of Programming Languages*, 2001.

[16] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the Symposium on Principles of Programming Languages*, 2007.

[17] P. Godefroid, A. Kiezun, and M. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005.

[19] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2008.

[20] I. Jager and D. Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab, Feb. 2010.

[21] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[22] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the ACM Conference on Software Engineering*, pages 416–426, 2007.

[23] G. C. Necula. Proof-carrying code. In *Proceedings of the Symposium on Principles of Programming Languages*, 1997.

[24] G. Nelson. A generalization of dijkstra's calculus. 11(4):517–561, 1989.

[25] C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *STTT*, 11(4):339–353, 2009.

[26] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution of binary programs. In *International Symposium on Software Testing and Analysis*, 2009.

[27] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering*, 2005.

[28] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *International Symposium on Software Testing and Analysis*, 2004.