Directed Proof Generation for Machine Code*

A. Thakur[1], J. Lim[1], A. Lal[2], A. Burton[1],
E. Driscoll[1], M. Elder[1]**, T. Andersen[1], and T. Reps[1,3]***

[1] University of Wisconsin; Madison, WI, USA
[2] Microsoft Research India; Bangalore, India
[3] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. We present the algorithms used in McVeto (**M**achine-**C**ode **VE**rification **TO**ol), a tool to check whether a stripped machine-code program satisfies a safety property. The verification problem that McVeto addresses is challenging because it cannot assume that it has access to (i) certain structures commonly relied on by source-code verification tools, such as control-flow graphs and call-graphs, and (ii) meta-data, such as information about variables, types, and aliasing. It cannot even rely on out-of-scope local variables and return addresses being protected from the program's actions. What distinguishes McVeto from other work on software model checking is that it shows how verification of machine-code can be performed, while avoiding conventional techniques that would be unsound if applied at the machine-code level.

1 Introduction

Recent research has led to new kinds of tools for analyzing programs for bugs and security vulnerabilities. In these tools, program analysis conservatively answers the question "Can the program reach a bad state?" Many impressive results have been achieved; however, the vast majority of existing tools analyze source code, whereas most programs are delivered as machine code. If analysts wish to vet such programs for bugs and security vulnerabilities, tools for analyzing machine code are needed.

Machine-code analysis presents many new challenges. For instance, at the machine-code level, memory is one large byte-addressable array, and an analyzer must handle computed—and possibly non-aligned—addresses. It is crucial to track array accesses and updates accurately; however, the task is complicated by the fact that arithmetic and dereferencing operations are both pervasive and inextricably intermingled. For instance, if local variable `x` is at offset –12 from the activation record's frame pointer (register `ebp`), an access on `x` would be turned

* Supported, in part, by NSF under grants CCF-{0540955, 0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 09-1-0776}, by ARL under grant W911NF-09-1-0413, and by AFRL under grant FA9550-09-1-0279.
** Supported by an NSF Graduate Fellowship.
*** T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

into an operand [ebp–12]. Evaluating the operand first involves pointer arithmetic ("ebp–12") and then dereferencing the computed address ("[·]"). On the other hand, machine-code analysis also offers new opportunities, in particular, the opportunity to track low-level, platform-specific details, such as memory-layout effects. Programmers are typically unaware of such details; however, they are often the source of exploitable security vulnerabilities.

The algorithms used in software model checkers that work on source code [5, 23, 6] would be be unsound if applied to machine code. For instance, before starting the verification process proper, SLAM [5] and BLAST [23] perform flow-insensitive (and possibly field-sensitive) points-to analysis. However, such analyses often make unsound assumptions, such as assuming that the result of an arithmetic operation on a pointer always remains inside the pointer's original target. Such an approach assumes—without checking—that the program is ANSI C compliant, and hence causes the model checker to ignore behaviors that are allowed by some compilers (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of structs or arrays, and are subsequently dereferenced). A program can use such features for good reasons—e.g., as a way for a C program to simulate subclassing [36]—but they can also be a source of bugs and security vulnerabilities.

This paper presents the techniques that we have implemented in a model checker for machine code, called MCVETO (**M**achine-**C**ode **VE**rification **TO**ol). MCVETO uses *directed proof generation* (DPG) [21] to find either an input that causes a (bad) target state to be reached, or a proof that the bad state cannot be reached. (The third possibility is that MCVETO fails to terminate.)

What distinguishes the work on MCVETO is that it addresses a large number of issues that have been ignored in previous work on software model checking, and would cause previous techniques to be unsound if applied to machine code. The contributions of our work can be summarized as follows:

1. We show how to verify safety properties of machine code while avoiding a host of assumptions that are unsound in general, and that would be inappropriate in the machine-code context, such as reliance on symbol-table, debugging, or type information, and preprocessing steps for (a) building a precomputed, fixed, interprocedural control-flow graph (ICFG), or (b) performing points-to/alias analysis.
2. MCVETO builds its (sound) abstraction of the program's state space on-the-fly, performing disassembly one instruction at a time during state-space exploration, without static knowledge of the split between code vs. data. (It does not have to be prepared to disassemble *collections* of nested branches, loops, procedures, or the whole program all at once, which is what can confuse conventional disassemblers [28].)

   The initial abstraction has only two abstract states, defined by the predicates "PC = *target*" and "PC $\neq$ *target*" (where "PC" denotes the program counter). The abstraction is gradually refined as more of the program is

exercised (§3). MCVETO can analyze programs with instruction aliasing[4] because it builds its abstraction of the program's state space entirely on-the-fly (§3.1). Moreover, MCVETO is capable of verifying (or detecting flaws in) self-modifying code (SMC). With SMC there is no fixed association between an address and the instruction at that address, but this is handled automatically by MCVETO's mechanisms for abstraction refinement. To the best of our knowledge, MCVETO is the first model checker to handle SMC.

3. MCVETO introduces *trace generalization*, a new technique for eliminating *families* of infeasible traces (§3.1). Compared to prior techniques that also have this ability [7, 22], our technique involves *no calls on an SMT solver*, and *avoids the potentially expensive step of automaton complementation*.
4. MCVETO introduces a new approach to performing DPG on multi-procedure programs (§3.3). Godefroid et al. [20] presented a declarative framework that codifies the mechanisms used for DPG in SYNERGY [21], DASH [6], and SMASH [20] (which are all instances of the framework). In their framework, *inter*procedural DPG is performed by invoking *intra*procedural DPG as a subroutine. In contrast, MCVETO's algorithm lies outside of that framework: the interprocedural component of MCVETO uses (and refines) an *infinite graph*, which is finitely represented and queried by *symbolic operations*.
5. We developed a language-independent algorithm to identify the aliasing condition relevant to a property in a given state (§3.4). Unlike previous techniques [6], it applies when static names for variables/objects are unavailable.

Items 1 and 2 address execution details that are typically ignored (unsoundly) by source-code analyzers. Items 3, 4, and 5 are applicable to both source-code and machine-code analysis. MCVETO is not restricted to an impoverished language. In particular, it handles pointers and bit-vector arithmetic.

In our implementation, we restricted ourselves to use only language-independent techniques. In particular, we used a technique for generating automatically some of the key primitives of MCVETO's analysis components from a description of an instruction set's semantics [27, 26]—i.e., (a) an emulator for running tests, (b) a primitive for performing symbolic execution, and (c) a primitive for the pre-image operator (Pre). In addition, we developed language-independent approaches to the issues discussed above (e.g., item 5). Consequently, our system acts as a Yacc-like tool for creating versions of MCVETO for different instruction sets: given an instruction-set description, a version of MCVETO is generated automatically. We created two such instantiations of MCVETO from descriptions of the Intel x86 and PowerPC instruction sets.

Organization. §2 contains a brief review of DPG. §3 describes the new DPG techniques used in MCVETO. §4 describes how different instances of MCVETO are generated automatically from a specification of the semantics of an instruction set. §5 presents experimental results. §6 discusses related work. §7 concludes.

[4] Programs written in instruction sets with varying-length instructions, such as x86, can have "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream [28].

2 Background on Directed Proof Generation (DPG)

Given a program P and a particular control location *target* in P, DPG returns either an input for which execution leads to *target* or a proof that *target* is unreachable (or DPG does not terminate). Two approximations of P's state space are maintained:

- A set T of concrete traces, obtained by running P with specific inputs. T *under*approximates P's state space.
- A graph G, called the *abstract graph*, obtained from P via abstraction (and abstraction refinement). G *over*approximates P's state space.

Nodes in G are labeled with formulas; edges are labeled with program statements or program conditions. One node is the *start node* (where execution begins); another node is the *target node* (the goal to reach). Information to relate the under- and overapproximations is also maintained: a concrete state σ in a trace in T is called a *witness* for a node n in G if σ satisfies the formula that labels n.

If G has no path from *start* to *target*, then DPG has proved that *target* is unreachable, and G serves as the proof. Otherwise, DPG locates a *frontier*: a triple (n, I, m), where (n, m) is an edge on a path from *start* to *target* such that n has a witness w but m does not, and I is the instruction on (n, m). DPG either performs concrete execution (attempting to reach *target*) or refines G by splitting nodes and removing certain edges (which may prove that *target* is unreachable). Which action to perform is determined using the basic step from *directed test generation* [18], which uses symbolic execution to try to find an input that allows execution to cross frontier (n, I, m). Symbolic execution is performed over symbolic states, which have two components: a *path constraint*, which represents a constraint on the input state, and a *symbolic map*, which represents the current state in terms of input-state quantities. DPG performs symbolic execution along the path taken during the concrete execution that produced witness w for n; it then symbolically executes I, and conjoins to the path constraint the formula obtained by evaluating m's predicate ψ with respect to the symbolic map. It calls an SMT solver to determine if the path constraint obtained in this way is satisfiable. If so, the result is a satisfying assignment that is used to add a new execution trace to T. If not, DPG refines G by splitting node n into n' and n'', as shown in Fig. 1.

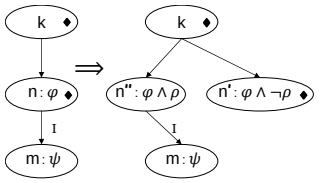


Fig. 1. The general refinement step across frontier (n, I, m). The presence of a witness is indicated by a "♦" inside of a node.

Refinement changes G to represent some *non-connectivity* information: in particular, n' is not connected to m in the refined graph (see Fig. 1). Let ψ be the formula that labels m, c be the concrete witness of n, and S_n be the symbolic state obtained from the symbolic execution up to n. DPG chooses a formula ρ, called the *refinement predicate*, and splits node n into n' and n'' to distinguish the cases when n is reached with a concrete state that satisfies ρ (n'') and when it is reached with a state that satisfies $\neg\rho$ (n'). The predicate ρ is chosen such that (i) no state that satisfies $\neg\rho$ can lead to a state that satisfies ψ after the

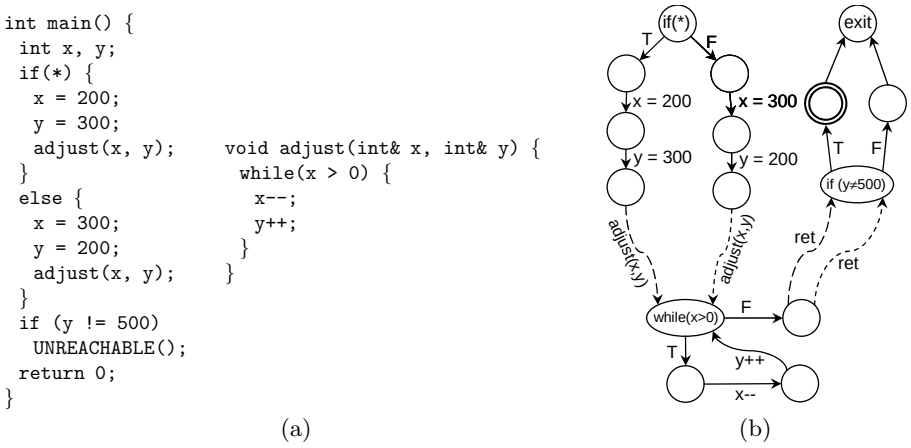


Fig. 2. (a) A program with a non-deterministic branch; (b) the program's ICFG.

execution of I, and (ii) the symbolic state S_n satisfies $\neg\rho$. Condition (i) ensures that the edge from n' to m can be removed. Condition (ii) prohibits extending the current path along I (forcing the DPG search to explore different paths). It also ensures that c is a witness for n' and not for n'' (because c satisfies S_n)—and thus the frontier during the next iteration must be different.

3 MCVETO

This section explains the methods used to achieve contributions 1–5 listed in §1. While MCVETO was designed to provide sound DPG for machine code, a number of its novel features are also useful for source-code DPG. Thus, to make the paper more accessible, our running example is the C++ program in Fig. 2. It makes a non-deterministic choice between two blocks that each call procedure `adjust`, which loops—decrementing `x` and incrementing `y`. Note that the affine relation $x + y = 500$ holds at the two calls on `adjust`, the loop-head in `adjust`, and the branch on `y!=500`.

Representing the Abstract Graph. The infinite abstract graph used in MCVETO is finitely represented as a nested word automaton (NWA) [2] and queried by symbolic operations. (See App. A for definitions related to NWAs.) As discussed in §3.1 the key property of NWAs for abstraction refinement is that, even though they represent matched call/return structure, they are closed under intersection [2]. That is, given NWAs A_1 and A_2, one can construct an NWA A_3 such that $L(A_3) = L(A_1) \cap L(A_2)$.

In our NWAs, the alphabet consists of all possible machine-code instructions. In addition, we annotate each state with a predicate. Operations on NWAs extend cleanly to accommodate the semantics of predicates—e.g., the $\cap$ operation labels a product state $\langle q_1, q_2 \rangle$ with the conjunction of the predicates on states q_1 and q_2. In MCVETO's abstract graph, we treat the value of the PC as data; consequently, predicates can refer to the value of the PC (see Fig. 3).

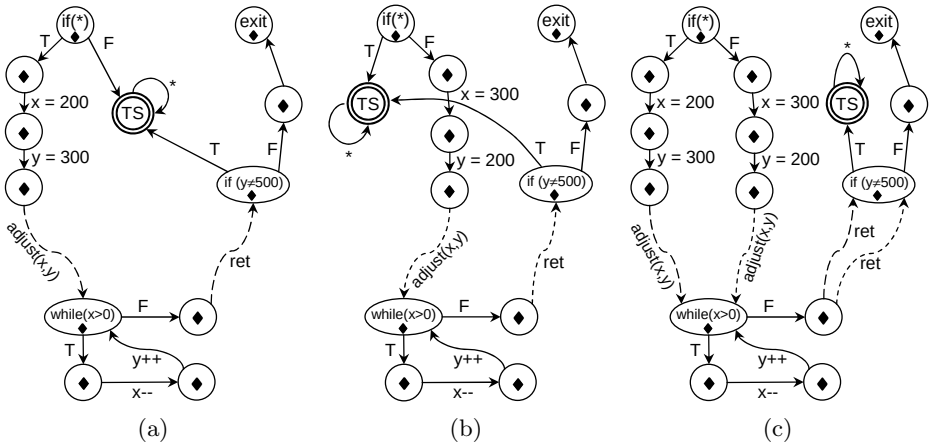


Fig. 4. (a) and (b) Two generalized traces, each of which reaches the end of the program. (c) The intersection of the two generalized traces. (A "♦" indicates that a node has a witness.)

3.1 Abstraction Refinement Via Trace Generalization

In a source-code model checker, the initial overapproximation of a program's state space is often the program's ICFG. Unfortunately, for machine code it is difficult to create an accurate ICFG *a priori* because of the use of indirect jumps, jump tables, and indirect function calls—as well as more esoteric features, such as instruction aliasing and SMC. For this reason, MCVETO begins with the degenerate NWA-based abstract graph G_0 shown in Fig. 3, which overapproximates the program's state space; i.e., G_o accepts an overapproximation of the set of minimal[5] traces that reach *target*. The abstract graph is refined during the state-space exploration carried out by MCVETO.

To avoid having to disassemble collections of nested branches, loops, procedures, or the whole program all at once, MCVETO performs *trace-based disassembly*: as concrete traces are generated during DPG, instructions are disassembled one at a time by decoding the current bytes of memory starting at the value of the PC. Each indirect jump or indirect call encountered can be resolved to a specific address. Trace-based disassembly is one of the techniques that allows MCVETO to handle self-modifying code.

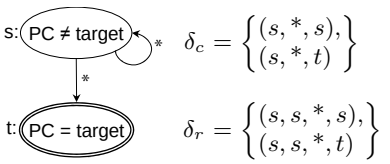


Fig. 3. (a) Internal-transitions in the initial NWA-based abstract graph G_0 created by MCVETO; (b) call- and return-transitions in G_0. * is a wild-card symbol that matches all instructions.

MCVETO uses each concrete trace $\pi \in T$ to refine abstract graph G. As mentioned in §2, the set T of concrete traces *under*approximates the program's state space, whereas G represents an

[5] A trace τ that reaches *target* is *minimal* if τ does not have a proper prefix that reaches *target*.

*over*approximation of the state space. MCVETO repeatedly solves instances of the following *trace-generalization* problem:

> Given a trace π, which is an *under*approximation of the program, convert π into an NWA-based abstract graph G_π that is an *over*approximation of the program.

We create G_π by "folding" π—grouping together all nodes with the same PC value, and augmenting it in a way that overapproximates the portion of the program not explored by π (denoted by $\pi/[\text{PC}]$); see Figs. 4(a) and (b) and Fig. 5. In particular, G_π contains one accepting state, called *TS* (for "target surrogate"). *TS* is an accepting state because it represents *target*, as well as all non-*target* locations not visited by π.

We now have two overapproximations, the original abstract graph G and folded trace G_π. Thus, by performing $G := G \cap G_\pi$, information about the portion of the program explored by π is incorporated into G, producing a third, improved overapproximation; see Fig. 4(c). (Equivalently, intersection eliminates the family of infeasible traces represented by the complement of G_π; however, because we already have G_π in hand, no automaton-complementation operation is required—cf. [22].)

The issue of how one forms an NWPrefix from an instruction sequence—i.e., identifying the nesting structure—is handled by a policy in the trace-recovery tool for classifying each position as an internal-, call-, or return-position. Currently, for reasons discussed in §3.5, we use the following policy: the position of any form of `call` instruction is a call-position; the position of any form of `ret` instruction is a return-position. In essence, MCVETO uses `call` and `ret` instructions to restrict the instruction sequences considered. If these match the program's actual instruction sequences, we obtain the benefits of the NWA-based approach—especially the reuse of information among refinements of a given procedure. The basic MCVETO algorithm is stated as Alg. 1.

Trace Generalization for Self-Modifying Code. To perform trace generalization for self-modifying code, state names are now of the form $q_{a,I}$, where a is an address and I is an instruction. Item 1 of Fig. 5 is changed to

- All positions $1 \leq k < |w| + 1$ for which (i) $\text{PC}[k]$ has a given address a, and (ii) $w[k]$ has a given instruction I are collapsed to a single NWA state $q_{a,I}$. All such states are rejecting states (the target was not reached).

Internal-, call-, and return-steps are now quadruples, $\langle \text{PC}[i], w[i], \text{PC}[i+1], w[i+1] \rangle$ depending on whether i, for $1 \leq i < |w|$, is an internal-, call-, or return-position, respectively. Other items are changed accordingly to account for instructions in state names. In addition, items [8]–[10] are replaced by

- For each position i, $1 \leq i < |w| + 1$, G_π contains
  - an internal-transition $(q_{\text{PC}[i],w[i]}, *, TS)$
  - a call-transition $(q_{\text{PC}[i],w[i]}, *, TS)$
  - a return-transition $(q_{\text{PC}[i],w[i]}, q_{\text{PC}[j],w[j]}, *, TS)$, where $1 \leq j < i$ and $w[j]$ is the unmatched `call` with largest index in $w[1..i-1]$.

Definition 1. *A trace π that does not reach target is represented by (i) a nested-word prefix $(w, \rightsquigarrow)$ over instructions (App. A), together with (ii) an array of PC values, $PC[1..|w|+1]$, where $PC[|w|+1]$ has the special value HALT if the trace terminated execution.* **Internal-steps**, **call-steps**, *and* **return-steps** *are triples of the form $\langle PC[i], w[i], PC[i+1]\rangle$, $1 \le i < |w|$, depending on whether i is an internal-position, call-position, or return-position, respectively. Given π, we construct $G_\pi \stackrel{\text{def}}{=} \pi/[PC]$ as follows:*

1. *All positions $1 \le k < |w|+1$ for which $PC[k]$ has a given address a are collapsed to a single NWA state q_a. All such states are rejecting states (the target was not reached).*
2. *For each internal-step $\langle a, I, b\rangle$, G_π has an internal-transition (q_a, I, q_b).*
3. *For each call-step $\langle a_c, \texttt{call}, a_e\rangle$, G_π has a call-transition $(q_{a_c}, \texttt{call}, q_{a_e})$. ("`call`" stands for whatever instruction instance was used in the call-step.)*
4. *For each return-step $\langle a_x, \texttt{ret}, a_r\rangle$ for which the PC at the call predecessor holds address a_c, G_π has a return-transition $(q_{a_x}, q_{a_c}, \texttt{ret}, q_{a_r})$. ("`ret`" stands for whatever instruction instance was used in the return-step.)*
5. *G_π contains one accepting state, called TS (for "target surrogate"). TS is an accepting state because it represents target, as well as all the non-target locations that π did not explore.*
6. *G_π contains three "self-loops": $(TS, *, TS) \in \delta_i$, $(TS, *, TS) \in \delta_c$, and $(TS, TS, *, TS) \in \delta_r$. (We use "*" in the latter two transitions because there are many forms of `call` and `ret` instructions.)*
7. *For each unmatched instance of a call-step $\langle a_c, \texttt{call}, a_e\rangle$, G_π has a return-transition $(TS, q_{a_c}, *, TS)$. (We use * because any kind of `ret` instruction could appear in the matching return-step.)*
8. *Let B_b denote a (direct or indirect) branch that takes branch-direction b.*
   - *If π has an internal-step $\langle a, B_b, c\rangle$ but not an internal-step $\langle a, B_{\neg b}, d\rangle$, G_π has an internal-transition $(q_a, B_{\neg b}, TS)$.*
   - *For each internal-step $\langle a, B_T, c\rangle$, where B is an indirect branch, G_π has an internal-transition (q_a, B_T, TS).*
9. *For each call-step $\langle a_c, \texttt{call}, a_e\rangle$ where `call` is an indirect call, G_π has a call-transition $(q_{a_c}, \texttt{call}, TS)$.*
10. *If $PC[|w|+1] \neq HALT$, G_π has an internal-transition $(q_{PC[|w|]}, I, TS)$, where "I" stands for whatever instruction instance was used in step $|w|$ of π. (We assume that an uncompleted trace never stops just before a `call` or `ret`.)*
11. *If $PC[|w|+1] = HALT$, G_π has an internal-transition $(q_{PC[|w|]}, I, Exit)$, where "I" stands for whatever instruction instance was used in step $|w|$ of π and Exit is a distinguished non-accepting state.*

Fig. 5. Definition of the trace-folding operation $\pi/[\text{PC}]$.

3.2 Speculative Trace Refinement

Motivated by the observation that DPG is able to avoid exhaustive loop unrolling if it discovers the right loop invariant, we developed mechanisms to discover candidate invariants from a folded trace, which are then incorporated into the abstract graph via NWA intersection. Although they are only *candidate* invariants, they are introduced into the abstract graph in the hope that they are invariants for the full program. The basic idea is to apply dataflow analysis to a graph obtained from the folded trace G_π. The recovery of invariants from G_π

Algorithm 1 Basic MCVETO algorithm (including trace-based disassembly)

```
 1: π := nested-word prefix for an execution run on a random initial state
 2: T := {π}; G_π := π/[PC]; G := (NWA from Fig. 3) ∩ G_π
 3: loop
 4:    if target has a witness in T then return "reachable"
 5:    Find a path τ in G from start to target
 6:    if no path exists then return "not reachable"
 7:    Find a frontier (n, I, m) in G, where concrete state σ witnesses n
 8:    Perform symbolic execution of the instructions of the concrete trace that reaches
       σ, and then of instruction I; conjoin to the path constraint the formula obtained
       by evaluating m's predicate ψ with respect to the symbolic map; let S be the
       path constraint so obtained
 9:    if S is feasible, with satisfying assignment A then
10:       π := nested-word prefix for an execution run on A
11:       T := T ∪ {π}; G_π := π/[PC]; G := G ∩ G_π
12:    else
13:       Refine G along frontier (n, I, m) (see Fig. 1)
```

is similar in spirit to the computation of invariants from traces in Daikon [15], but in MCVETO they are computed *ex post facto* by dataflow analysis on the folded trace. While any kind of dataflow analysis could be used in this fashion, MCVETO currently uses two analyses:

- Affine-relation analysis [30] is used to obtain linear equalities over registers and a set of memory locations, V. V is computed by running aggregate structure identification [32] on G_π to obtain a set of inferred memory variables M, then selecting $V \subseteq M$ as the most frequently accessed locations in π.
- An analysis based on strided-interval arithmetic [33] is used to discover range and congruence constraints on the values of individual registers and memory locations.

The candidate invariants are used to create predicates for the nodes of G_π. Because an analysis may not account for the full effects of indirect memory references on the inferred variables, to incorporate a discovered candidate invariant φ for node n into G_π safely, we split n on φ and $\neg\varphi$. Again we have two overapproximations: G_π, from the folded trace, augmented with the candidate invariants, and the original abstract graph G. To incorporate the candidate invariants into G, we perform $G := G \cap G_\pi$; the $\cap$ operation labels a product state $\langle q_1, q_2 \rangle$ with the conjunction of the predicates on states q_1 of G and q_2 of G_π.

Fig. 6 shows how the candidate affine relation $\varphi \stackrel{\text{def}}{=} x + y = 500$ would be introduced at the loop-head of `adjust` in the generalized traces from Figs. 4(a) and (b). (Relation φ does, in fact, hold for the portions of the state space explored by Figs. 4(a) and (b).) With this enhancement, subsequent steps of DPG will be able to show that the dotted loop-heads (labeled with $\neg\varphi$) can never be reached from *start*. In addition, the predicate φ on the solid loop-heads enables DPG to avoid exhaustive loop unrolling to show that the true branch of `y!=500` can never be taken.

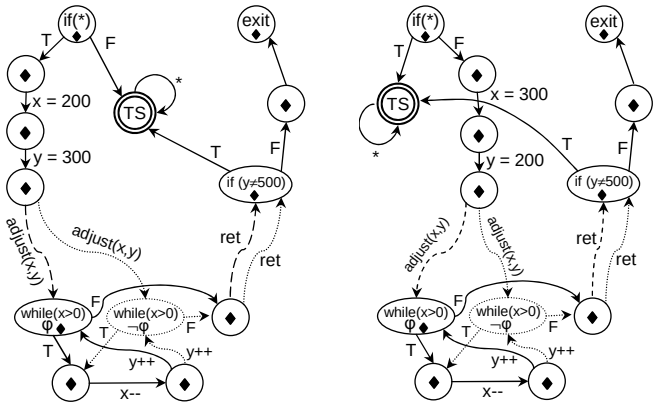


Fig. 6. Fig. 4(a) and (b) with the loop-head in `adjust` split with respect to the candidate invariant $\varphi \stackrel{\text{def}}{=} x + y = 500$.

```
int y;
void baz(){
  y=0;
  y++;
  y--;
}
```

```
void lotsaBaz(int a){
  y=0;
  if(a>0) baz();
  if(a>1) baz();
  if(a>2) baz();
  if(a>3) baz();
  if(y!=0)
    ERR: return;
}
```

(a)

```
int bar1() {
  int i,r = 0;
  for(i=0;i<100;i++){
    complicated(); r++;
  }
  return r;
}

int bar2(){ return 10; }
```

```
void foo(int x){
  int y;
  if(x == 0) y = bar2();
  else y = bar1();
  if(y == 10)
    ERR: return;
}
```

(b)

Fig. 7. Programs that illustrate the benefit of using a conceptually infinite abstract graph.

3.3 Symbolic Methods for Interprocedural DPG

In other DPG systems [21, 6, 20], *inter*procedural DPG is performed by invoking *intra*procedural DPG as a subroutine. In contrast, MCVETO analyzes a representation of the entire program (refined on-the-fly), which allows it to reuse all information from previous refinement steps. For instance, in the program shown in Fig. 7(a), procedure `lotsaBaz` makes several calls to `baz`. By invoking analysis once for each call site on `baz`, a tool such as DASH has to re-learn that `y` is set to 0. In contrast, MCVETO only needs to learn this once and gets automatic reuse at all call sites. Note that such reuse is achieved in a different way in SMASH [20], which makes use of explicit procedure summaries. However, because the split between local and global variables is not known when analyzing machine code, it is not clear to us how MCVETO could generate such explicit summaries.

Furthermore, SMASH is still restricted to invoking intraprocedural analysis as a subroutine, whereas MCVETO is not limited to considering frontiers in just a single procedure: at each stage, it is free to choose a frontier in *any* procedure. To see why such freedom can be important, consider the source-code example in Fig. 7(b) (where *target* is `ERR`). DASH might proceed as follows. The initial

test uses [x ↦ 42], which goes through bar1, but does not reach *target*. After a few iterations, the frontier is the call to bar1, at which point DASH is invoked on bar1 to prove that the return value is not 10. The subproof takes a long time because of the complicated loop in bar1. In essence, DASH gets stuck in bar1 without recourse to an easier way to reach *target*. MCVETO can make the same choices, and would start to prove the same property for the return value of bar1. However, refinements inside of bar1 cause the abstract graph to grow, and at some point, if the policy is to pick a frontier *closest* to *target*, the frontier switches to one in main that is closer to *target*—in particular, the true branch of the if-condition x==0. MCVETO will be able to extend that frontier by running a test with [x ↦ 0], which will go through bar2 and reach *target*. The challenge that we face to support such flexibility is how to select the frontier while accounting for paths that reflect the nesting structure of calls and returns. As discussed below, by doing computations via automata, transducers, and pushdown systems, MCVETO can find the set of *all* frontiers, as well as identify the k *closest* frontiers.

Symbolic Methods to Find All Frontiers and Closest Frontiers. The first step is to convert the abstract graph from an NWA to a PDS to be able to use standard symbolic reachability queries on the PDS [8, 16].

Definition 2. *A **pushdown system** (PDS) is a four-tuple $\mathcal{P} = (P, Act, \Gamma, \Delta)$, where P is a finite set of **control locations**, Act is a finite set of **actions**, Γ is a finite set of **stack symbols**, and $\Delta \subseteq P \times \Gamma \times Act \times P \times \Gamma^*$ is a finite set of **rules**. A **configuration** of $\mathcal{P}$ is a pair $\langle p, u\rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma\rangle \overset{\sigma}{\hookrightarrow} \langle p', u\rangle$, where $p, p' \in P$, $\sigma \in Act$, $\gamma \in \Gamma$, and $u \in \Gamma^*$. For $\sigma \in Act$, the rules define a collection of **transition relations** $\overset{\sigma}{\Longrightarrow}$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma\rangle \overset{\sigma}{\hookrightarrow} \langle p', u'\rangle$, then $\langle p, \gamma u\rangle \overset{\sigma}{\Longrightarrow} \langle p', u'u\rangle$ for all $u \in \Gamma^*$. For a word $\sigma_1 \ldots \sigma_n \in Act^*$, the relation $\overset{\sigma_1 \ldots \sigma_n}{\Longrightarrow}$ is defined in the obvious way.*

To convert an NWA to a PDS, each NWA transition is converted to one or more PDS rules:

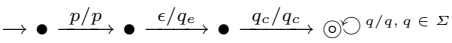
- For each transition $(q, \sigma, q') \in \delta_i$, the PDS has a rule $\langle p, q\rangle \overset{\sigma}{\hookrightarrow} \langle p, q'\rangle$.
- For each transition $(q_c, \sigma, q_e) \in \delta_c$, the PDS has a rule $\langle p, q_c\rangle \overset{\sigma}{\hookrightarrow} \langle p, q_e\, q_c\rangle$.
- For each transition $(q_x, q_c, \sigma, q_r) \in \delta_r$, the PDS has two rules, $\langle p, q_x\rangle \overset{\epsilon}{\hookrightarrow} \langle p_x, \epsilon\rangle$ and $\langle p_x, q_c\rangle \overset{\sigma}{\hookrightarrow} \langle p, q_r\rangle$.

We often work with abstractions of PDSs in which action symbols are dropped: $\Rightarrow \overset{\text{def}}{=} \bigcup_{\sigma \in Act} \overset{\sigma}{\Longrightarrow}$. Let $\Rightarrow^*$ denote the reflexive transitive closure of $\Rightarrow$. For a set of configurations C, $pre^*(C) \overset{\text{def}}{=} \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) \overset{\text{def}}{=} \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$—i.e., backward and forward reachability, respectively, with respect to transition relation $\Rightarrow$. When C is a regular language of configurations, automata for the configuration languages $pre^*(C)$ and $post^*(C)$ can be computed in polynomial time. Using the PDS constructed from the NWA-based abstract graph, line [1] of Alg. 1 can be performed by testing whether $start \in pre^*(target)$.

Given the set of all $\langle \text{PC}, \mathit{stack} \rangle$ configurations for concrete states that occur in some execution trace, let X be the corresponding set of configurations in $P \times \Gamma^*$ for the PDS of the abstract graph. It is straightforward to build an automaton that recognizes X because we can recover $\langle \text{PC}, \mathit{stack} \rangle$ information during a traversal of an NWPrefix. (Henceforth, the automaton is referred to as X, as well.) We also create an automaton InChopButNotExecuted for the part of the chop between *start* and *target* that has not been reached during a concrete execution:

$$\text{InChopButNotExecuted} = \mathit{post}^*(\mathit{start}) \cap \mathit{pre}^*(\mathit{target}) \cap \neg X.$$

Given an automaton A for a language $L(A)$, let ID[A] be the transducer for the projection of the identity relation on $L(A)$: $\{(a, a) \mid a \in L(A)\}$. It is straightforward to create a transducer for the *post* relation on PDS configurations: e.g., $\langle p, q_c \rangle \hookrightarrow \langle p, q_e\, q_c \rangle$ contributes the fragment



to the transducer. (Note that the transducer encodes *post*, not *post**.) Now we put these together to find all frontiers:

$$\text{Frontiers} = \text{ID}[X] \circ \mathit{post} \circ \text{ID}[\text{InChopButNotExecuted}],$$

where $\circ$ denotes transducer composition. In English, what this does is the following: Frontiers identifies—as a relation between configuration pairs of the form (x, icbne)—all edges in the infinite transition relation of the abstract graph in which

1. x is reached during concrete execution (ID[X])
2. one can go from x to *icbne* in one step (*post*)
3. *icbne* is on a path from *start* to *target* in the infinite transition relation of the abstract graphs PDS, but was not reached during a concrete execution (ID[InChopButNotExecuted])

The composition with the two projection functions, "ID[X] $\circ \ldots$" and "$\ldots \circ$ ID[InChopButNotExecuted]", serves to specialize *post* to just the edges in the infinite transition relation of the abstract graph that run between a configuration in X and a configuration in InChopButNotExecuted.

We can obtain "closest frontiers" by using *weighted* PDSs [34, 9] and adding shortest-distance weights to either $\mathit{pre}^*(\mathit{target})$ (to obtain frontiers that are closest to *target*) or to $\mathit{post}^*(\mathit{start})$ (to obtain frontiers that are closest to *start*), and then carrying the weights through the transducer-composition operations.

3.4 A Language-Independent Approach to Aliasing Relevant to a Property

This section describes how MCVETO identifies—in a language-independent way suitable for use with machine code—the aliasing condition relevant to a property in a given state. Lim et al. showed how to generate a Pre primitive for machine code [26]; however, repeated application of Pre causes refinement predicates to

explode. We now present a language-independent algorithm for obtaining an aliasing condition α that is suitable for use in machine-code analysis. From α, one immediately obtains Pre_α. There are two challenges to defining an appropriate notion of aliasing condition for use with machine code: (i) `int`-valued and address-valued quantities are indistinguishable at runtime, and (ii) arithmetic on addresses is used extensively.

Suppose that the frontier is (n, I, m), ψ is the formula on m, and S_n is the symbolic state obtained via symbolic execution of a concrete trace that reaches n. For source code, Beckman et al. [6] identify aliasing condition α by looking at the relationship, in S_n, between the addresses written to by I and the ones used in ψ. However, their algorithm for computing α is language-*dependent*: their algorithm has the semantics of C implicitly encoded in its search for "the addresses written to by I". In contrast, as explained below, we developed an alternative, language-*independent* approach, both to identifying α and computing Pre_α.

For the moment, to simplify the discussion, suppose that a concrete machine-code state is represented using two maps $M : \mathit{INT} \to \mathit{INT}$ and $R : \mathit{REG} \to \mathit{INT}$. Map M represents memory, and map R represents the values of machine registers. (A more realistic definition of memory is considered later in this section.) We use the standard theory of arrays to describe (functional) updates and accesses on maps, e.g., $\mathit{update}(m, k, d)$ denotes the map m with index k updated with the value d, and $\mathit{access}(m, k)$ is the value stored at index k in m. (We use the notation $m(r)$ as a shorthand for $\mathit{access}(m, r)$.) We also use the standard axiom from the theory of arrays: $(\mathit{update}(m, k_1, d))(k_2) = \mathit{ite}(k_1 = k_2, d, m(k_2))$, where *ite* is an *if-then-else* term. Suppose that I is "`mov [eax],5`" (which corresponds to `*eax = 5` in source-code notation) and that ψ is $(M(R(\texttt{ebp}) - 8) + M(R(\texttt{ebp}) - 12) = 10)$.[6] First, we symbolically execute I starting from the identity symbolic state $S_{id} = [M \mapsto M, R \mapsto R]$ to obtain the symbolic state $S' = [M \mapsto \mathit{update}(M, R(\texttt{eax}), 5), R \mapsto R]$. Next, we evaluate ψ under S'—i.e., perform the substitution $\psi[M \leftarrow S'(M), R \leftarrow S'(R)]$. For instance, the term $M(R(\texttt{ebp}) - 8)$, which denotes the contents of memory at address $R(\texttt{ebp}) - 8$, evaluates to $(\mathit{update}(M, R(\texttt{eax}), 5))(R(\texttt{ebp}) - 8)$. From the axiom for arrays, this simplifies to $\mathit{ite}(R(\texttt{eax}) = R(\texttt{ebp}) - 8, 5, M(R(\texttt{ebp}) - 8))$. Thus, the evaluation of ψ under S' yields

$$\begin{pmatrix} \mathit{ite}(R(\texttt{eax}) = R(\texttt{ebp}) - 8, 5, M(R(\texttt{ebp}) - 8)) \\ + \ \mathit{ite}(R(\texttt{eax}) = R(\texttt{ebp}) - 12, 5, M(R(\texttt{ebp}) - 12)) \end{pmatrix} = 10 \qquad (1)$$

This formula equals $\text{Pre}(I, \psi)$ [26].

The process described above illustrates a general property: for any instruction I and formula ψ, $\text{Pre}(I, \psi) = \psi[M \leftarrow S'(M), R \leftarrow S'(R)]$, where $S' = \text{SE}[\![I]\!]S_{id}$ and $\text{SE}[\![\cdot]\!]$ denotes symbolic execution [26].

The next steps are to identify α and to create a simplified formula ψ' that weakens $\text{Pre}(I, \psi)$. These are carried out simultaneously during a traversal of $\text{Pre}(I, \psi)$ that makes use of the symbolic state S_n at node n. We illustrate this

[6] In x86, `ebp` is the frame pointer, so if program variable `x` is at offset –8 and `y` is at offset –12, ψ corresponds to $\texttt{x} + \texttt{y} = 10$.

on the example discussed above for a case in which $S_n(R) = [\texttt{eax} \mapsto R(\texttt{ebp}) - 8]$ (i.e., continuing the scenario from footnote 6, `eax` holds `&x`). Because the *ite*-terms in Eqn. (1) were generated from array accesses, *ite*-conditions represent possible constituents of aliasing conditions. We initialize α to *true* and traverse Eqn. (1). For each subterm t of the form $ite(\varphi, t_1, t_2)$ where φ definitely holds in symbolic state S_n, t is simplified to t_1 and φ is conjoined to α. If φ can never hold in S_n, t is simplified to t_2 and $\neg\varphi$ is conjoined to α. If φ can sometimes hold and sometimes fail to hold in S_n, t and α are left unchanged.

In our example, $R(\texttt{eax})$ equals $R(\texttt{ebp}) - 8$ in symbolic state S_n; hence, applying the process described above to Eqn. (1) yields

$$\begin{array}{rcl} \psi' & = & (5 + M(R(\texttt{ebp}) - 12) = 10) \\ \alpha & = & (R(\texttt{eax}) = R(\texttt{ebp}) - 8) \wedge (R(\texttt{eax}) \neq R(\texttt{ebp}) - 12) \end{array} \tag{2}$$

The formula $\alpha \Rightarrow \psi'$ is the desired refinement predicate $\mathrm{Pre}_\alpha(I, \psi)$.

In practice, we found it beneficial to use an alternative approach, which is to perform the same process of evaluating conditions of *ite* terms in $\mathrm{Pre}(I, \psi)$, but to use one of the concrete witness states W_n of frontier node n in place of symbolic state S_n. The latter method is less expensive (it uses formula-evaluation steps in place of SMT solver calls), but generates an aliasing condition specific to W_n rather than one that covers all concrete states described by S_n.

Both approaches are *language-independent* because they isolate where the instruction-set semantics comes into play in $\mathrm{Pre}(I, \psi)$ to the computation of $S' = \mathrm{SE}[\![I]\!]S_{id}$; all remaining steps involve only purely logical primitives. Although our algorithm computes $\mathrm{Pre}(I, \psi)$ explicitly, that step alone does not cause an explosion in formula size; explosion is due to *repeated* application of Pre. In our approach, the formula obtained via $\mathrm{Pre}(I, \psi)$ is immediately simplified to create first ψ', and then $\alpha \Rightarrow \psi'$.

Byte-Addressable Memory. We assumed above that the memory map has type $INT \rightarrow INT$. When memory is byte-addressable, the actual memory-map type is $INT32 \rightarrow INT8$. This complicates matters because accessing (updating) a 32-bit quantity in memory translates into four contiguous 8-bit accesses (updates). For instance, a 32-bit little-endian access can be expressed as follows:

$$\begin{array}{rcl} access\_32\_8\_LE\_32(m, a) & = & \text{let } v4 = 2^{24} * Int8To32ZE(m(a+3)) \\ & & \quad\; v3 = 2^{16} * Int8To32ZE(m(a+2)) \\ & & \quad\; v2 = 2^{8} * Int8To32ZE(m(a+1)) \\ & & \quad\; v1 = Int8To32ZE(m(a)) \\ & & \text{in } (v4 \mid v3 \mid v2 \mid v1) \end{array} \tag{3}$$

where *Int8To32ZE* converts an *INT8* to an *INT32* by padding the high-order bits with zeros, and "|" denotes bitwise-or.

Let *update*_32_8_*LE*_32 denote the similar operation for updating a map of type $INT32 \rightarrow INT8$ under the little-endian storage convention. Note that when $1 \leq |k_1 -_{INT32} k_2| \leq 3$, we no longer have the property

$$access\_32\_8\_LE\_32(update\_32\_8\_LE\_32(M, k_1, d), k_2) = access\_32\_8\_LE\_32(M, k_2).$$

$$
\begin{array}{l}
\left(\begin{array}{l}
\ 2^{24} * \mathit{Int8To32ZE}(\mathit{ite}(x+3=p+3,0,\mathit{ite}(x+3=p+2,0,\mathit{ite}(x+3=p+1,0,\mathit{ite}(x+3=p,5,*(x+3))))))\\
|\ 2^{16} * \mathit{Int8To32ZE}(\mathit{ite}(x+2=p+3,0,\mathit{ite}(x+2=p+2,0,\mathit{ite}(x+2=p+1,0,\mathit{ite}(x+2=p,5,*(x+2))))))\\
|\ 2^{8} * \mathit{Int8To32ZE}(\mathit{ite}(x+1=p+3,0,\mathit{ite}(x+1=p+2,0,\mathit{ite}(x+1=p+1,0,\mathit{ite}(x+1=p,5,*(x+1))))))\\
|\ \mathit{Int8To32ZE}(\mathit{ite}(x=p+3,0,\mathit{ite}(x=p+2,0,\mathit{ite}(x=p+1,0,\mathit{ite}(x=p,5,*x)))))
\end{array}\right)\\
+\left(\begin{array}{l}
\ 2^{24} * \mathit{Int8To32ZE}(\mathit{ite}(y+3=p+3,0,\mathit{ite}(y+3=p+2,0,\mathit{ite}(y+3=p+1,0,\mathit{ite}(y+3=p,5,*(y+3))))))\\
|\ 2^{16} * \mathit{Int8To32ZE}(\mathit{ite}(y+2=p+3,0,\mathit{ite}(y+2=p+2,0,\mathit{ite}(y+2=p+1,0,\mathit{ite}(y+2=p,5,*(y+2))))))\\
|\ 2^{8} * \mathit{Int8To32ZE}(\mathit{ite}(y+1=p+3,0,\mathit{ite}(y+1=p+2,0,\mathit{ite}(y+1=p+1,0,\mathit{ite}(y+1=p,5,*(y+1))))))\\
|\ \mathit{Int8To32ZE}(\mathit{ite}(y=p+3,0,\mathit{ite}(y=p+2,0,\mathit{ite}(y=p+1,0,\mathit{ite}(y=p,5,*y)))))
\end{array}\right)\\
=10
\end{array}
$$

Fig. 8. The formula for $\mathrm{Pre}(I,\psi)$, where ψ is $\mathit{update\_32\_8\_LE\_32}(M, R(\texttt{ebp})-8) + \mathit{update\_32\_8\_LE\_32}(M, R(\texttt{ebp})-12) = 10$, obtained by evaluating ψ on the symbolic state $S' = [M \mapsto \mathit{update\_32\_8\_LE\_32}(M, R(\texttt{eax}), 5), R \mapsto R]$. For brevity, the following notational shorthands are used in the formula: $p = R(\texttt{eax})$, $x = R(\texttt{ebp})-8$, $y = R(\texttt{ebp})-12$, $*x = M(R(\texttt{ebp})-8)$, $*y = M(R(\texttt{ebp})-12)$, etc.

and hence it is invalid to simplify formulas by the rule

$$\begin{array}{l}\mathit{access\_32\_8\_LE\_32}(\mathit{update\_32\_8\_LE\_32}(M,k_1,d),k_2)\\ \qquad\Rightarrow\ \mathit{ite}(k_1=k_2,d,\mathit{access\_32\_8\_LE\_32}(M,k_2)).\end{array}$$

However, the four single-byte accesses on m in Eqn. (3) ($m(a)$, $m(a+1)$, $m(a+2)$, and $m(a+3)$) are *access* operations for which it is valid to apply the standard axiom of arrays (i.e., $(m[k_1 \mapsto d])(k_2) = \mathit{ite}(k_1 = k_2, d, m(k_2))$).

Returning to the example discussed above, in which $R(\texttt{eax})$ equals $R(\texttt{ebp})-8$ in symbolic state S_n, we perform the same steps as before. First, the symbolic execution of $I =$ `mov [eax],5` starting from the identity symbolic state $S_{id} = [M \mapsto M, R \mapsto R]$ results in the symbolic state

$$S' = [M \mapsto \mathit{update\_32\_8\_LE\_32}(M, R(\texttt{eax}), 5), R \mapsto R].$$

The formula ψ is now written as follows:

$$\mathit{access\_32\_8\_LE\_32}(M, R(\texttt{ebp})-8) + \mathit{access\_32\_8\_LE\_32}(M, R(\texttt{ebp})-12) = 10.$$

To obtain $\mathrm{Pre}(I,\psi)$, we evaluate ψ under S', which yields the formula shown in Fig. 8.

The formula shown in Fig. 8 is the analog of Eqn. (1).

The step that uses symbolic state S_n to identify α and create a simplified formula ψ' that weakens $\mathrm{Pre}(I,\psi)$ is now applied to the formula shown in Fig. 8 and produces

$$\psi' \stackrel{\text{def}}{=} 5 + \left(\begin{array}{l}\ 2^{24} * \mathit{Int8To32ZE}(*(y+3))\\ |\ 2^{16} * \mathit{Int8To32ZE}(*(y+2))\\ |\ 2^{8} * \mathit{Int8To32ZE}(*(y+1))\\ |\ \mathit{Int8To32ZE}(*y)\end{array}\right) = 10.$$

The α that is the analog of Eqn. (2) is the conjunction of the disequalities collected from the formula shown in Fig. 8:

$$\begin{array}{l}\alpha \stackrel{\text{def}}{=} x+3 \neq p+3 \wedge \ldots x+3 \neq p \wedge \ldots x \neq p+3 \wedge \ldots x \neq p\\ \quad \wedge\ y+3 \neq p+3 \wedge \ldots y+3 \neq p \wedge \ldots y \neq p+3 \wedge \ldots y \neq p.\end{array}$$

As before, the formula $\alpha \Rightarrow \psi'$ is the desired refinement predicate $\mathrm{Pre}_\alpha(I,\psi)$.

3.5 Soundness Guarantee

The soundness argument for MCVETO is more subtle than it otherwise might appear because of examples like the one shown in Fig. 9. The statement `*(&r+2) = r;` overwrites `foo`'s return address, and `MakeChoice` returns a random 32-bit number. At the end of `foo`, half the runs return normally to `main`. For the other half, the `ret` instruction at the end of `foo` serves to call `bar`. The problem is that for a run that returns normally to `main` after trace generalization and intersection with G_0, there is no frontier. Consequently, half of the runs of MCVETO, on average, would erroneously report that location `ERR` is unreachable.

```
void bar() {
  ERR: // address here is 0x10
}

void foo() {
  int b = MakeChoice() & 1;
  int r = b*0x68 + (1-b)*0x10;
  *(&r+2) = r;
  return;
}

int main() {
  foo();
  // address here is 0x68
}
```

Fig. 9. `ERR` is reachable, but only along a path in which a `ret` instruction serves to perform a call.

MCVETO uses the following policy P for classifying execution steps: (a) the position of any form of `call` instruction is a call-position; (b) the position of any form of `ret` instruction is a return-position. Our goals are (i) to define a property Q that is compatible with P in the sense that MCVETO can check for violations of Q *while checking only NWPrefix paths* (App. A), and (ii) to establish a soundness guarantee: either MCVETO reports that Q is violated (along with an input that demonstrates it), or it reports that *target* is reachable (again with an input that demonstrates it), or it correctly reports that Q is invariant and *target* is unreachable. To define Q, we augment the instruction-set semantics with an auxiliary stack. Initially, the auxiliary stack is empty; at each `call`, a copy of the return address pushed on the processor stack is also pushed on the auxiliary stack; at each `ret`, the auxiliary stack is popped.

Definition 3. *An **acceptable execution** (AE) under the instrumented semantics is one in which at each `ret` instruction (i) the auxiliary stack is non-empty, and (ii) the address popped from the processor stack matches the address popped from the auxiliary stack.*

In the instrumented semantics, a flag V is set whenever the program performs an execution step that violates either condition (i) or (ii) of Defn. 3. Instead of the initial NWA shown in Fig. 3, we use a similar two-state NWA that has states q_1: PC $\neq$ *target* $\wedge \neg$V and q_2: PC $=$ *target* $\vee$ V, where q_1 is non-accepting and q_2 is accepting. In addition, we add one more rule to the trace-generalization construction for G_π from Fig. 5:

12. For each return-step $\langle a_x, \texttt{ret}, a_r \rangle$, G_π has an internal-transition $(q_{a_x}, \texttt{ret}, TS)$.

As shown below, these modifications cause the DPG algorithm to also search for traces that are AE violations.

Theorem 1 (Soundness of MCVETO).

1. *If McVeto reports "AE violation" (together with an input S), execution of S performs an execution that is not an AE.*
2. *If McVeto reports "bug found" (together with an input S), execution of S performs an AE to target.*
3. *If McVeto reports "OK", then (a) the program performs only AEs, and (b) target cannot be reached during any AE.*

Sketch of Proof: If a program has a concrete execution trace that is not AE, there must exist a shortest non-AE prefix, which has the form "NWPrefix `ret`" where either (i) the auxiliary stack is empty, or (ii) the return address used by `ret` from the processor stack fails to match the return address on the auxiliary stack. At each stage, the abstract graph used by MCVETO accepts an overapproximation of the program's shortest non-AE execution-trace prefixes. This is true of the initial graph G_0 because internal transitions have wild-card symbols. Moreover, each folded trace $G_\pi = \pi/[\text{PC}]$ accepts traces of the form "NWPrefix `ret`" due to the addition of internal transitions to *TS* for each `ret` instruction (item 12 above). NWA intersection of two sound overapproximations leads to a refined sound overapproximation. Therefore, when MCVETO has shown that no accepting state is reachable, it has also proved that the program has no AE violations.

For an example like Fig. 9, MCVETO reports "AE violation".

In cases when MCVETO reports "AE violation", it can indicate a stack-smashing attack. If one wishes to find out more information when there is an AE violation, one can run a purely intraprocedural version of MCVETO that does not give special treatment to `call` and `ret` instructions. This is potentially more expensive than running the interprocedural version of MCVETO, but can find out additional information about executions that are not AE.

4 Implementation

The MCVETO implementation incorporates all of the techniques described in §3. The implementation uses only language-independent techniques; consequently, MCVETO can be easily retargeted to different languages. The main components of MCVETO are language-independent in two different dimensions:

1. The MCVETO DPG driver is structured so that one only needs to provide implementations of primitives for concrete and symbolic execution of a language's constructs, plus a handful of other primitives (e.g., Pre_α). Consequently, this component can be used for both source-level languages and machine-code languages.
2. For machine-code languages, we used two tools that *generate* the required implementations of the primitives for concrete and symbolic execution from descriptions of the syntax and concrete operational semantics of an instruction set. The abstract syntax and concrete semantics are specified using TSL (**T**ransformer **S**pecification **L**anguage) [27]. Translation of binary-encoded instructions to abstract syntax trees is specified using a tool called ISAL

(**I**nstruction **S**et **A**rchitecture **L**anguage).[7] The relationship between ISAL and TSL is similar to the relationship between Flex and Bison—i.e., a Flex-generated lexer passes tokens to a Bison-generated parser. In our case, the TSL-defined abstract syntax serves as the formalism for communicating values—namely, instructions' abstract syntax trees—between the two tools.

In addition, we developed language-independent solutions to each of the issues in MCVETO, such as identifying the aliasing condition relevant to a specific property in a given state (§3.4). Consequently, our implementation acts as a Yacc-like tool for creating versions of MCVETO for different languages: given a description of language L, a version of MCVETO for L is generated automatically. We created two specific instantiations of MCVETO from descriptions of the Intel x86 and PowerPC instruction sets. To perform symbolic queries on the conceptually-infinite abstract graph (§3.3), the implementation uses OpenFst [1] (for transducers) and WALi [24] (for WPDSs).

5 Experiments

Our experiments (see Fig. 10) were run on a single core of a single-processor quad-core 3.0 GHz Xeon computer running Windows XP, configured so that a user process has 4 GB of memory. They were designed to test various aspects of a DPG algorithm and to handle various intricacies that arise in machine code (some of which are not visible in source code). We compiled the programs with Visual Studio 8.0, and ran MCVETO on the resulting object files (without using symbol-table information).[8]

The examples `ex5`, `ex6`, and `ex8` are from the NECLA Static Analysis Benchmarks.[9] The examples `barber`, `berkeley`, `cars`, `efm` are multi-procedure versions of the larger examples on which SYNERGY [21] was tested. (SYNERGY was tested using single-procedure versions only.[10]) `Instraliasing` illustrates the ability to handle instruction aliasing. (The instruction count for this example was obtained via static disassembly, and hence is only approximate.) `Smc1` illustrates the ability of MCVETO to handle self-modifying code. `Underflow` is taken from a DHS tutorial on security vulnerabilities. It illustrates a `strncpy` vulnerability.

The examples are small, but challenging. They demonstrate MCVETO's ability to reason automatically about low-level details of machine code using a sequence of sound abstractions. The question of whether the cost of soundness is inherent, or whether there is some way that the well-behavedness of (most) code could be exploited to make the analysis scale better is left for future research.

[7] ISAL also handles other kinds of concrete syntactic issues, including (a) *encoding* (abstract syntax trees to binary-encoded instructions), (b) *parsing assembly* (assembly code to abstract syntax trees), and (c) *assembly pretty-printing* (abstract syntax trees to assembly code).

[8] The examples are available at `www.cs.wisc.edu/wpis/examples/McVeto`.

[9] `www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php`

[10] `www.cse.iitb.ac.in/~bhargav/synergy`

Program		McVeto performance (x86)				
Name	Outcome	#Instrs	CE	SE	Ref	time
blast2/blast2	timeout	98	**	**	**	**
fib/fib–REACH-0	timeout	49	**	**	**	**
fib/fib–REACH-1	counterex.	49	1	0	0	0.125
slam1/slam1	proof	84	15	129	307	203
smc1/smc1–REACH-0*	proof	21	1	60	188	959
smc1/smc1–REACH-1*	counterex.	21	1	0	0	0.016
ex5/ex	counterex.	48	2	10	38	3.05
doubleloopdep/count–COUNT-6	counterex.	31	7	11	13	11.5
doubleloopdep/count–COUNT-7	counterex.	31	7	11	13	11.6
doubleloopdep/count–COUNT-8	counterex.	31	7	11	13	11.6
doubleloopdep/count–COUNT-9	counterex.	31	7	11	13	11.7
inter.synergy/barber	timeout	253	**	**	**	**
inter.synergy/berkeley	counterex.	104	5	13	16	3.95
inter.synergy/cars	proof	196	11	118	349	188
inter.synergy/efm	timeout	188	**	**	**	**
share/share–CASE-0	proof	50	3	24	75	8.5
cert/underflow	counterex.	120	2	6	12	9.55
instraliasing/instraliasing–REACH-0	proof	46	2	36	126	15.0
instraliasing/instraliasing–REACH-1	counterex.	46	2	17	55	5.86
longjmp/jmp	AE viol.	74	1	0	0	0.015
overview0/overview	proof	49	2	31	91	54.9
small_static_bench/ex5	proof	33	3	7	13	2.27
small_static_bench/ex6	proof	30	1	55	146	153
small_static_bench/ex8	proof	89	4	17	46	6.31
verisec-gxine/simp_bad	counterex.	1067	1	0	0	0.094
verisec-gxine/simp_ok	proof	1068	**	**	**	**
clobber_ret_addr/clobber–CASE-4	AE viol.	43	4	9	18	2.13
clobber_ret_addr/clobber–CASE-8	AE viol.	35	2	2	5	0.625
clobber_ret_addr/clobber–CASE-9	proof	35	1	5	21	1.44

Fig. 10. McVeto experiments. The columns show whether McVeto returned a proof, counterexample, or an AE violation (Outcome); the number of instructions (#Instrs); the number of concrete executions (CE); the number of symbolic executions (SE), which also equals the number of calls to the Yices solver; the number of refinements (Ref), which also equals the number of Pre_α computations; and the total time (in seconds). *SMC test case. **Exceeded twenty-minute time limit.

6 Related Work

Machine-Code Analyzers Targeted at Finding Vulnerabilities. A substantial amount of work exists on techniques to detect security vulnerabilities by analyzing source code for a variety of languages [38, 29, 39]. Less work exists on vulnerability detection for machine code. Kruegel et al. [25] developed a system for automating mimicry attacks; it uses symbolic execution to discover attacks that can give up and regain execution control by modifying the contents of the data, heap, or stack so that the application is forced to return control to injected attack code at some point after the execution of a system call. Cova et al. [14] used that platform to detect security vulnerabilities in x86 executables via symbolic execution.

Prior work exists on directed *test* generation for machine code [19, 10]. Directed test generation combines concrete execution and symbolic execution to find inputs that increase test coverage. An SMT solver is used to obtain inputs that force previously unexplored branch directions to be taken. In contrast,

MCVETO implements directed *proof* generation. Unlike directed-test-generation tools, MCVETO is goal-directed, and works by trying to refute the claim "no path exists that connects program entry to a given goal state".

Machine-Code Model Checkers. SYNERGY applies to an x86 executable for a "single-procedure C program with only [`int`-valued] variables" [21] (i.e., no pointers). It uses debugging information to obtain information about variables and types, and uses Vulcan [37] to obtain a CFG. It uses integer arithmetic—not bit-vector arithmetic—in its solver. Quoting A. Nori, "[[21] handles] the complexities of binaries via its front-end Vulcan and *not* via its property-checking engine" [31]. In contrast, MCVETO addresses the challenges of checking properties of stripped executables articulated in §1.

AIR ("Assembly Iterative Refinement") [13] is a model checker for PowerPC. AIR decompiles an assembly program to C, and then checks if the resulting C program satisfies the desired property by applying COPPER [12], a predicate-abstraction-based model checker for C source code. They state that the choice of COPPER is not essential, and that any other C model checker, such as SLAM [5] or BLAST [23] would be satisfactory. However, the C programs that result from their translation step use pointer arithmetic and pointer dereferencing, whereas—as mentioned in §1—many C model checkers, including SLAM and BLAST, make unsound assumptions about pointer arithmetic.

[MC]SQUARE [35] is a model checker for microcontroller assembly code. It uses explicit-state model-checking techniques (combined with a degree of abstraction) to check CTL properties.

Our group developed two prior machine-code model checkers, CodeSurfer/x86 [4] and DDA/x86 [3]. Neither system uses either under-approximation or symbolic execution. For overapproximation, both use numeric static analysis and a different form of abstraction refinement.

Self-Modifying Code. The work on MCVETO addresses a problem that has been almost entirely ignored by the PL research community. There is a paper on SMC by Gerth [17], and a recent paper by Cai et al. [11]. However, both of the papers concern proof systems for reasoning about SMC. In contrast, MCVETO can verify (or detect flaws in) SMC automatically. As far as we know, MCVETO is the first model checker to address verifying (or detecting flaws in) SMC.

Trace Generalization. The trace-generalization technique of §3.1 has both similarities to and differences from the *path programs* of Beyer et al. [7] and the *trace-refinement* technique of Heizmann et al. [22]. All three techniques refine an overapproximation to eliminate *families* of infeasible concrete traces. However, trace generalization obtains the desired outcome in a substantially different way. Beyer et al. analyze refuted abstract traces to obtain new predicates to refine the predicate abstraction in use. The subsequent refinement step requires possibly expensive calls on an SMT solver to compute new abstract transformers. Heizmann et al. adopt a language-theoretic viewpoint: once a refutation automaton is constructed—which involves calling an SMT solver and an interpolant generator—refinement is performed by automaton complementation followed by automaton intersection. In contrast, our generalized traces are created

by generalizing a *feasible concrete trace* to create directly a representation that overapproximates the set of minimal traces that reach *target*. Consequently, refinement by trace generalization involves *no calls on an SMT solver*, and *avoids the potentially expensive step of automaton complementation*.

7 Conclusion

MCVETO resolves many issues that have been unsoundly ignored in previous work on software model checking. MCVETO addresses the challenge of establishing properties of the machine code that actually executes, and thus provides one approach to checking the effects of compilation and optimization on correctness. The contributions of the paper lie in the insights that went into defining the innovations in dynamic and symbolic analysis used in MCVETO: (i) sound disassembly and sound construction of an overapproximation (even in the presence of instruction aliasing and self-modifying code) (§3.1), (ii) a new method to eliminate families of infeasible traces (§3.1), (iii) a method to speculatively, but soundly, elaborate the abstraction in use (§3.2), (iv) new symbolic methods to query the (conceptually infinite) abstract graph (§3.3), and (v) a language-independent approach to Pre_α (§3.4). Not only are our techniques language-independent, the implementation is parameterized by specifications of an instruction set's semantics. By this means, MCVETO has been instantiated for both x86 and PowerPC.

References

1. C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *CIAA*, 2007.
2. R. Alur and P. Madhusudan. Adding nesting structure to words. *JACM*, 56, 2009.
3. G. Balakrishnan and T. Reps. Analyzing stripped device-driver executables. In *TACAS*, 2008.
4. G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *CAV*, 2005.
5. T. Ball and S. Rajamani. The SLAM toolkit. In *CAV*, 2001.
6. N. Beckman, A. Nori, S. Rajamani, and R. Simmons. Proofs from tests. In *ISSTA*, 2008.
7. D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, 2007.
8. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR*, 1997.
9. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.
10. D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 2008.
11. H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *PLDI*, 2007.
12. S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *FMSD*, 25(2–3), 2004.

13. S. Chaki and J. Ivers. Software model checking without source code. In *Proc. of the First NASA Formal Methods Symposium*, 2009.
14. M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *ACSAC*, 2006.
15. M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *SCP*, 69, 2007.
16. A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *ENTCS*, 9, 1997.
17. R. Gerth. Formal verification of self modifying code. In *Int. Conf. for Young Computer Scientists*, 1991.
18. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.
19. P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
20. P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, 2010.
21. B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. SYNERGY: A new algorithm for property checking. In *FSE*, 2006.
22. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.
23. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
24. N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.
25. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Sec. Symp.*, 2005.
26. J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. In *SPIN Workshop*, 2009.
27. J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.
28. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS*, 2003.
29. B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Sec. Symp.*, 2005.
30. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.
31. A. Nori. Personal communication, Jan. 2009.
32. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, 1999.
33. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM*, 2006.
34. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58, 2005.
35. B. Schlich. *Model Checking of Software for Microcontrollers*. PhD thesis, RWTH Aachen University, Germany, 2008.
36. M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *FSE*, pages 180–198, 1999.
37. A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. MSR-TR-2001-50, Microsoft Research, Apr. 2001.
38. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, Feb. 2000.
39. Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Sec. Symp.*, 2006.

A Nested Words and Nested Word Automata

Definition 4 ([2]). *A **nested word** $(w, \rightsquigarrow)$ over alphabet Σ is an ordinary word $w \in \Sigma^*$, together with a **nesting relation** $\rightsquigarrow$ of length $|w|$. $\rightsquigarrow$ is a collection of edges (over the positions in w) that do not cross. A nesting relation of length $l \geq 0$ is a subset of $\{-\infty, 1, 2, \ldots, l\} \times \{1, 2, \ldots, l, +\infty\}$ such that*

- *Nesting edges only go forwards: if $i \rightsquigarrow j$ then $i < j$.*
- *No two edges share a position: for $1 \leq i \leq l$, $|\{j \mid i \rightsquigarrow j\}| \leq 1$ and $|\{j \mid j \rightsquigarrow i\}| \leq 1$.*
- *Edges do not cross: if $i \rightsquigarrow j$ and $i' \rightsquigarrow j'$, then one cannot have $i < i' \leq j < j'$.*

*When $i \rightsquigarrow j$ holds, for $1 \leq i \leq l$, i is called a **call** position; if $i \rightsquigarrow +\infty$, then i is a **pending call**; otherwise i is a **matched call**, and the unique position j such that $i \rightsquigarrow j$ is called its **return successor**. Similarly, when $i \rightsquigarrow j$ holds, for $1 \leq j \leq l$, j is a **return** position; if $-\infty \rightsquigarrow j$, then j is a **pending return**, otherwise j is a **matched return**, and the unique position i such that $i \rightsquigarrow j$ is called its **call predecessor**. A position $1 \leq i \leq l$ that is neither a call nor a return is an **internal** position.*

***MatchedNW** denotes the set of nested words that have no pending calls or returns. **NWPrefix** denotes the set of nested words that have no pending returns.*

*A **nested word automaton** (NWA) A is a 5-tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and δ is a transition relation. The transition relation δ consists of three components, $(\delta_c, \delta_i, \delta_r)$, where*

- *$\delta_i \subseteq Q \times \Sigma \times Q$ is the transition relation for internal positions.*
- *$\delta_c \subseteq Q \times \Sigma \times Q$ is the transition relation for call positions.*
- *$\delta_r \subseteq Q \times Q \times \Sigma \times Q$ is the transition relation for return positions.*

*Starting from q_0, an NWA A reads a nested word $nw = (w, \rightsquigarrow)$ from left to right, and performs transitions (possibly non-deterministically) according to the input symbol and $\rightsquigarrow$. If A is in state q when reading input symbol σ at position i in w, and i is an internal or call position, A makes a transition to q' using $(q, \sigma, q') \in \delta_i$ or $(q, \sigma, q') \in \delta_c$, respectively. If i is a return position, let k be the call predecessor of i, and q_c be the state A was in just before the transition it made on the k^{th} symbol; A uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q'. If, after reading nw, A is in a state $q \in F$, then A **accepts** nw.*