There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code* (Invited Tutorial)

T. Reps[1,2]**, J. Lim[1], A. Thakur[1], G. Balakrishnan[3], and A. Lal[4]

[1] University of Wisconsin; Madison, WI, USA
[2] GrammaTech, Inc.; Ithaca, NY, USA
[3] NEC Laboratories America, Inc.; Princeton, NJ, USA
[4] Microsoft Research India; Bangalore, India

Abstract. This paper discusses the obstacles that stand in the way of doing a good job of machine-code analysis. Compared with analysis of source code, the challenge is to drop all assumptions about having certain kinds of information available (variables, control-flow graph, call-graph, etc.) and also to address new kinds of behaviors (arithmetic on addresses, jumps to "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream, self-modifying code, etc.).

The paper describes some of the challenges that arise when analyzing machine code, and what can be done about them. It also provides a rationale for some of the design decisions made in the machine-code-analysis tools that we have built over the past few years.

1 Introduction

This paper is intended to complement the papers that we have written over the past few years on verifying safety properties of stripped executables. Elsewhere (e.g., [9] and [3, §1]) we have argued at length the benefits of analyzing machine code rather than source code. In brief,

- Machine code is an artifact that is closer to what actually executes on the machine; models derived from machine code can be more accurate than models derived from source code (particularly because compilation, optimization, and link-time transformation can change how the code behaves).
- When source code is compiled, the compiler and optimizer make certain choices that eliminate some possible behaviors—hence there is sometimes the opportunity to obtain more precise answers from machine-code analysis than from source-code analysis.

* 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.

** T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology discussed in this publication.

Rather than rehashing those arguments here, we take them as givens, and focus instead on the obstacles standing in the way of doing a good job of machine-code analysis. The paper explains some of the challenges that arise when analyzing machine code, and what can be done about them. It thereby provides a rationale for some of the design decisions made in the machine-code-analysis tools that we have built over the past few years, in particular CodeSurfer/x86 [8, 3], DDA/x86 [7], and MCVETO [27]. Those three tools represent several firsts:

– CodeSurfer/x86 is the first program-slicing tool for machine code that is able to track the flow of values through memory, and thus help with understanding dependences transmitted via memory loads and stores.
– DDA/x86 is the first automatic program-verification tool that is able to check whether a stripped executable—such as a device driver—conforms to an API-usage rule (specified as a finite-state machine).
– MCVETO is the first automatic program-verification tool capable of verifying (or detecting flaws in) self-modifying code.

As with any verification tool, each of these tools comes with a few caveats about the class of programs to which it can be applied, which are due to certain design decisions concerning the analysis techniques used.

The remainder of the paper is organized as follows: §2 describes some of the challenges presented by machine-code analysis and verification, as well as different aspects of the design space for analysis and verification tools. §3 discusses one point in the design space: when the goal is to account only for behaviors expected from a standard compilation model, but report evidence of possible deviations from such behaviors. §4 discusses another point in the design space: when the goal is to verify machine code, including accounting for deviant behaviors. §5 discusses how we have created a way to build "Yacc-like" tool generators for machine-code analysis and verification tools (i.e., from a semantic specification of a language L, we are able to create automatically an instantiation of a given tool for L). §6 concerns related work. (Portions of the paper are based on material published elsewhere, e.g., [7, 3, 21, 27].)

2 The Design Space for Machine-Code Analysis

Machine-code-analysis problems come in at least three varieties: (i) in addition to the executable, the program's source code is also available; (ii) the source code is unavailable, but the executable includes symbol-table/debugging information ("unstripped executables"); (iii) the executable has no symbol-table/debugging information ("stripped executables"). The appropriate variant to work with depends on the intended application. Some analysis techniques apply to multiple variants, but other techniques are severely hampered when symbol-table/debugging information is absent. In our work, we have primarily been concerned with the analysis of stripped executables, both because it is the most challenging situation, and because it is what is needed in the common situation where one needs to install a device driver or commercial off-the-shelf application delivered as stripped machine code. If an individual or company wishes

to vet such programs for bugs, security vulnerabilities, or malicious code (e.g., back doors, time bombs, or logic bombs) analysis tools for stripped executables are required.

Compared with source-code analysis, analysis of stripped executables presents several problems. In particular, standard approaches to source-code analysis assume that certain information is available—or at least obtainable by separate analysis phases with limited interactions between phases, e.g.,

- a control-flow graph (CFG), or interprocedural CFG (ICFG)
- a call-graph
- a set of variables, split into disjoint sets of local and global variables
- a set of non-overlapping procedures
- type information
- points-to information or alias information

The availability of such information permits the use of techniques that can greatly aid the analysis task. For instance, when one can assume that (i) the program's variables can be split into (a) global variables and (b) local variables that are encapsulated in a conceptually protected environment, and (ii) a procedure's return address is never corrupted, analyzers often tabulate and reuse explicit summaries that characterize a procedure's behavior.

Source-code-analysis tools sometimes also use questionable techniques, such as interpreting operations in integer arithmetic, rather than bit-vector arithmetic. They also usually make assumptions about the semantics that are not true at the machine-code level—for instance, they usually assume that the area of memory beyond the top-of-stack is not part of the execution state at all (i.e., they adopt the fiction that such memory does not exist).

In general, analysis of stripped executables presents many challenges and difficulties, including

absence of information about variables: In stripped executables, no information is provided about the program's global and local variables.

a semantics based on a flat memory model: With machine code, there is no notion of separate "protected" storage areas for the local variables of different procedure invocations, nor any notion of protected fields of an activation record. For instance, a procedure's return address is stored on the stack; an analyzer must prove that it is not corrupted, or discover what new values it could have.

absence of type information: In particular, int-valued and address-valued quantities are indistinguishable at runtime.

arithmetic on addresses is used extensively: Moreover, numeric and address-dereference operations are inextricably intertwined, even during simple operations. For instance, consider the load of a local variable `v`, located at offset `-12` in the current activation record, into register `eax`: `mov eax,[ebp-12]`.[5]

[5] For readers who need a brief introduction to the 32-bit Intel x86 instruction set (also called IA32), it has six 32-bit general-purpose registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, and `edi`), plus two additional registers: `ebp`, the frame pointer, and `esp`, the stack pointer. By convention, register `eax` is used to pass back the return value from

```
void foo() {
  int arr[2], n;
  void (*addr_bar)() = bar;
  if(MakeChoice() == 7) n = 4; // (*)
  else n = 2;
  for(int i = 0; i < n; i++)
    arr[i] = (int)addr_bar; // (**)
  return; // can return to the entry of bar
}
```

```
void bar() {
  ERR: return;
}

int main() {
  foo();
  return 0;
}
```

Fig. 1. A program that, on some executions, can modify the return address of `foo` so that `foo` returns to the beginning of `bar`, thereby reaching `ERR`. (`MakeChoice` is a primitive that returns a random 32-bit number.)

This instruction involves a *numeric* operation (`ebp-12`) to calculate an address whose value is then *dereferenced* (`[ebp-12]`) to fetch the value of `v`, after which the value is placed in `eax`.

instruction aliasing: 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 [22].

self-modifying code: With self-modifying code there is no fixed association between an address and the instruction at that address.

Because certain kinds of information ordinarily available during source-code analysis (variables, control-flow graph, call-graph, etc.) are not available when analyzing machine code, some standard techniques are precluded. For instance, source-code analysis tools often use separate phases of (i) points-to/alias analysis (analysis of addresses) and (ii) analysis of arithmetic operations. Because numeric and address-dereference operations are inextricably intertwined, as discussed above, only very imprecise information would result with the same organization of analysis phases.

Fig. 1 is an example that will be used to illustrate two points in the design space of machine-code-analysis tools with respect to the question of corruption of a procedure's return address. When the program shown in Fig. 1 is compiled with Visual Studio 2005, the return address is located two 4-byte words beyond `arr`—in essence, at `arr[3]`. When `MakeChoice` returns 7 at line (*), `n` is set to 4, and thus in the loop `arr[3]` is set to the starting address of procedure `bar`.

---

a function call. In Intel assembly syntax, the movement of data is from right to left (e.g., `mov eax,ecx` sets the value of `eax` to the value of `ecx`). Arithmetic and logical instructions are primarily two-address instructions (e.g., `add eax,ecx` performs `eax := eax + ecx`). An operand in square brackets denotes a dereference (e.g., if `v` is a local variable stored at offset -12 off the frame pointer, `mov [ebp-12],ecx` performs `v := ecx`). Branching is carried out according to the values of condition codes ("flags") set by an earlier instruction. For instance, to branch to `L1` when `eax` and `ebx` are equal, one performs `cmp eax,ebx`, which sets `ZF` (the zero flag) to 1 iff `eax` − `ebx` = 0. At a subsequent jump instruction `jz L1`, control is transferred to `L1` if `ZF` = 1; otherwise, control falls through.

Consequently, the execution of `foo` can modify `foo`'s return address so that `foo` returns to the beginning of `bar`.

In general, tools that represent different points in the design space have different answers to the question

> *What properties are checked, and what is expected of the analyzer after the first anomalous action is detected?*

First, consider the actions of a typical source-code analyzer, which would propagate abstract states through an interprocedural control-flow graph (ICFG). The call on `foo` in `main` causes it to begin analyzing `foo`. Once it is finished analyzing `foo`, it would follow the "return-edge" in the ICFG back to the point in `main` after the call on `foo`. However, a typical source-code analyzer does not represent the return address explicitly in the abstract state and relies on an unsound assumption that the return address cannot be modified. The analyzer would never analyze the path from `main` to `foo` to `bar`, and would thus miss one of the program's possible behaviors. The analyzer might report an array-out-of-bounds error at line (**).

As explained in more detail in §3, in CodeSurfer/x86 and DDA/x86, we were able to make our analysis problems resemble standard source-code analysis problems, to a considerable degree. One difference is that in CodeSurfer/x86 and DDA/x86 the return address is represented explicitly in the abstract state. At a return, the current (abstract) value of the return address is checked against the expected value. If the return address is not guaranteed to have the expected value, a report is issued that the return address may have been modified. However, for reasons explained in §3, the analyses used in CodeSurfer/x86 and DDA/x86 proceed according to the original return address—i.e., by returning from `foo` to `main`. Similar to source-code analyzers, they would not analyze the path from `main` to `foo` to `bar`. Although they miss one of the program's possible behaviors, they report that there is possibly an anomalous overwrite of the return address.

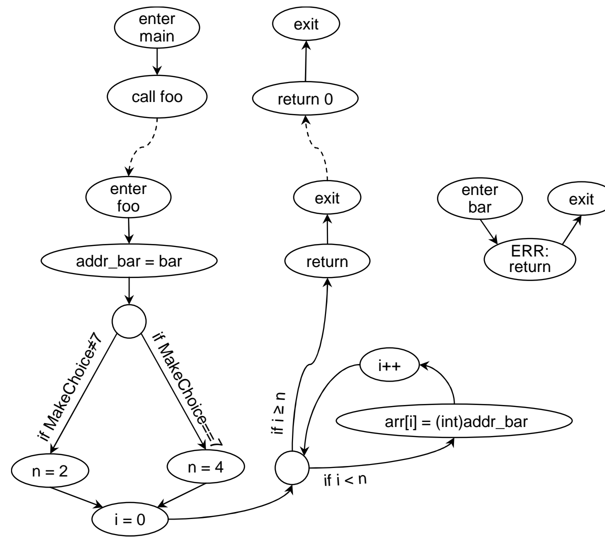

Fig. 2. Conventional ICFG for the program shown in Fig. 1. Note that the CFG for `bar` is disconnected from the rest of the ICFG.

In contrast, MCVETO uses some techniques that permit it not only to detect the presence of "deviant behaviors", but also to explore them as well. The state-space-exploration method used in MCVETO discovers that the execution of `foo` can modify `foo`'s return address. It uses the modified return address to discover that `foo` actually returns to the beginning of `bar`, and correctly reports that `ERR` is reachable.

3 Accounting for Behaviors Expected from a Standard Compilation Model

As illustrated in §2, CodeSurfer/x86[6] only follows behaviors expected from a standard compilation model. It is prepared to detect and report deviations from such behaviors, but not prepared to explore the consequences of deviant behavior. By a "standard compilation model", we mean that the executable has procedures, activation records (ARs), a global data region, and a free-storage pool; might use virtual functions and DLLs; maintains a runtime stack; each global variable resides at a fixed offset in memory; each local variable of a procedure f resides at a fixed offset in the ARs for f; actual parameters of f are pushed onto the stack by the caller so that the corresponding formal parameters reside at fixed offsets in the ARs for f; the program's instructions occupy a fixed area of memory, and are not self-modifying.

During the analyses performed by CodeSurfer/x86, these aspects of the program are checked. When violations are detected, an error report is issued, and the analysis proceeds. In doing so, however, we generally chose to have CodeSurfer/x86's analysis algorithms only explore behaviors that stay within those of the desired execution model. For instance, as discussed in §2, if the analysis discovers that the return address might be modified within a procedure, CodeSurfer/x86 reports the potential violation, but proceeds without modifying the control flow of the program. In the case of self-modifying code, either a write into the code will be reported or a jump or call to data will be reported.

If the executable conforms to the standard compilation model, CodeSurfer/x86 returns valid analysis results for it; if the executable does not conform to the model, then one or more violations will be discovered, and corresponding error reports will be issued; if the (human) analyst can determine that the error report is indeed a false positive, then the analysis results are valid. The advantages of this approach are three-fold: (i) it provides the ability to analyze some aspects of programs that may deviate from the desired execution model; (ii) it generates reports of possible deviations from the desired execution model; (iii) it does not force the analyzer to explore all of the consequences of each (apparent) deviation, which may be a false positive due to loss of precision that occurs during static analysis. If a deviation is possible, then at least one report will be a true positive: each possible *first* violation will be reported.

[6] Henceforth, we will not refer to DDA/x86 explicitly. Essentially all of the observations made about CodeSurfer/x86 apply to DDA/x86 as well.

Memory Model. Although in the concrete semantics of x86 machine code the activation records for procedures, the heap, and the memory area for global data are all part of one address space, for the purposes of analysis, CodeSurfer/x86 adopts an approach that is similar to that used in source-code analyzers: the address space is treated as being separated into a set of disjoint areas, which are referred to as *memory-regions*. Each memory-region represents a group of locations that have similar runtime properties; in particular, the runtime locations that belong to the ARs of a given procedure belong to one memory-region. Each (abstract) byte in a memory-region represents a set of concrete memory locations. For a given program, there are three kinds of regions: (1) the global-region, for memory locations that hold initialized and uninitialized global data, (2) AR-regions, each of which contains the locations of the ARs of a particular procedure, and (3) malloc-regions, each of which contains the locations allocated at a particular malloc site [5].

All data objects, whether local, global, or in the heap, are treated in a fashion similar to the way compilers arrange to access variables in local ARs, namely, via an offset: an abstract address in a memory-region is represented by a pair: (memory-region, offset). For an n-bit architecture, the size of each memory-region in the abstract memory model is 2^n. For each region, the range of offsets within the memory-region is $[-2^{n-1}, 2^{n-1}-1]$. Offset 0 in an AR-region represents all concrete starting addresses of the ARs that the AR-region represents. Offset 0 in a malloc-region represents all concrete starting addresses of the heap blocks that the malloc-region represents. Offset 0 of the global-region represents the concrete address 0. Nothing is assumed about the relative positions of memory-regions.

Analysis Algorithms. To a substantial degree, the analysis algorithms used in CodeSurfer/x86 closely resemble standard source-code analyses, although considerable work was necessary to map ideas from source-code analysis over to machine-code analysis. One of the main themes of the work on CodeSurfer/x86 was how an analyzer can bootstrap itself from preliminary intermediate representations (IRs) that record fairly basic information about the code of a stripped executable to IRs on which it is possible to run analyses that resemble standard source-code analyses. (See [3, §2.2,§4, and §5].)

The analyses used in CodeSurfer/x86 address the following problem:

> *Given a (possibly stripped) executable E, identify the procedures, data objects, types, and libraries that it uses, and,*
> – *for each instruction I in E and its libraries,*
> – *for each interprocedural calling context of I, and*
> – *for each machine register and variable V in scope at I,*
>
> *statically compute an accurate over-approximation to the set of values that V may contain when I executes.*

The work presented in our 2004 paper [4] provided a way to apply the tools of abstract interpretation [12] to the problem of analyzing stripped executables (using the memory model sketched above) to statically compute an over-approximation at each program point to the set of values that a register or memory location

could contain. We followed that work up with other techniques to complement and enhance the approach [26, 19, 25, 5, 6, 2, 7]. That body of work resulted in a method to recover a good approximation to an executable's variables and dynamically allocated memory objects, and to track the flow of values through them.

Caveats. Some of the limitations of CodeSurfer/x86 are due to the memory model that it uses. For instance, the memory-region-based memory model interferes with the ability to interpret masking operations applied to stack addresses. Rather than having *addr* & `MASK`, one has *(AR, offset)* & `MASK`, which generally results in $\top$ (i.e., any possible address) because nothing is known about the possible addresses of the base of AR, and hence nothing is known about the set of bit patterns that *(AR, offset)* represents. (Such masking operations are sometimes introduced by `gcc` to enforce stack alignment.)

4 Verification in the Presence of Deviant Behaviors

MCVETO has pioneered some techniques that permit it to verify safety properties of machine code, even if the program deviates from the behaviors expected from a standard compilation model. Because the goal is to account for deviant behaviors, the situation is more challenging than the one discussed in §3. For instance, in the case of self-modifying code, standard structures such as the ICFG and the call-graph are not even well-defined. That is, as discussed in §2, standard ways of interpreting the ICFG during analysis are not sound. One must look to other abstractions of the program's state space to accommodate such situations.

Our MCVETO tool addresses these issues by generalizing the source-code-analysis technique of directed proof generation (DPG) [16]. 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). DPG makes use of two approximations of P's state space:

- 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.

The two abstractions are played off one another, using the basic step from *directed test generation* [14] to determine whether it is possible to drive execution down a new path to *target*:

- If G has no path from *start* to *target*, then DPG has proven that *target* is unreachable, and G serves as the proof.
- If G has a path from *start* to *target* with a feasible prefix that has not been explored before, DPG initiates a concrete execution to attempt to reach *target*. Such a step augments the underapproximation T.
- If G has a path from *start* to *target* but the path has an infeasible prefix, DPG refines the overapproximation by performing the node-splitting operation shown in Fig. 3(a).

DPG is attractive for addressing the problem that we face, for two reasons. First, it is able to account for a program's deviant behaviors during the process

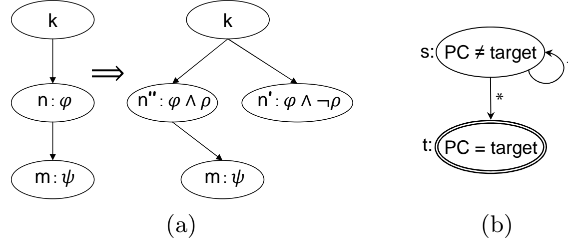

Fig. 3. (a) The general refinement step used in DPG. Refinement predicate ρ ensures that no execution can follow $n' \to m$. (b) The initial abstract graph used in McVeto. (* is a wild-card symbol that matches all instructions.)

of building up the underapproximation of the program's state space. Second, as we discuss below, the overapproximation of the program's state space can be constructed without relying on an ICFG or call-graph being available.

What Must be Handled Differently in Machine-Code DPG? The abstract graph used during DPG is an overapproximation of the program's state space. The versions of DPG used in SYNERGY [16], DASH [10], and SMASH [15] all start with an ICFG, which, when working with stripped machine code, is not only unavailable initially but may not even be well-defined. Nevertheless, for machine code, one can still create an over-approximation of the state space, as long as one makes a few adjustments to the basic elements of DPG.

1. The system needs to treat the value of the program counter (PC) as data so that predicates can refer to the value of the PC.
2. The system needs to learn the over-approximation starting with a cruder over-approximation than an ICFG. In particular, MCVETO starts from the initial abstraction shown in Fig. 3(b), which only has two abstract states, defined by the predicates "PC = *target*" and "PC $\neq$ *target*". The abstraction is gradually refined as more of the program is exercised.
3. To handle self-modifying code, a predicate that labels an abstract state may hold a constraint that specifies what instruction is decoded from memory, starting at the address held by the PC.
4. In addition to refinements of the abstract graph performed by the step shown in Fig. 3(a), the abstract graph is also refined each time a concrete execution fails to reach *target*. These refinements are inspired, in part, by the *trace-refinement* technique of Heizmann et al. [17]. The abstract graph is considered to be an automaton (e.g., s is a non-final state in Fig. 3(b), whereas target t is a final state). A concrete execution trace τ that reaches *target* is *minimal* if no proper prefix of τ reaches *target*. Each concrete execution trace that fails to reach *target* is *generalized* to create an automaton (or "folded trace") that accepts an overapproximation of the set of minimal concrete execution traces that reach *target*. The automaton is intersected with the current abstract graph to create the next version of the abstract graph.

The approach adopted by MCVETO has a number of advantages. First, it allows MCVETO to build a sound overapproximation of the program's state space

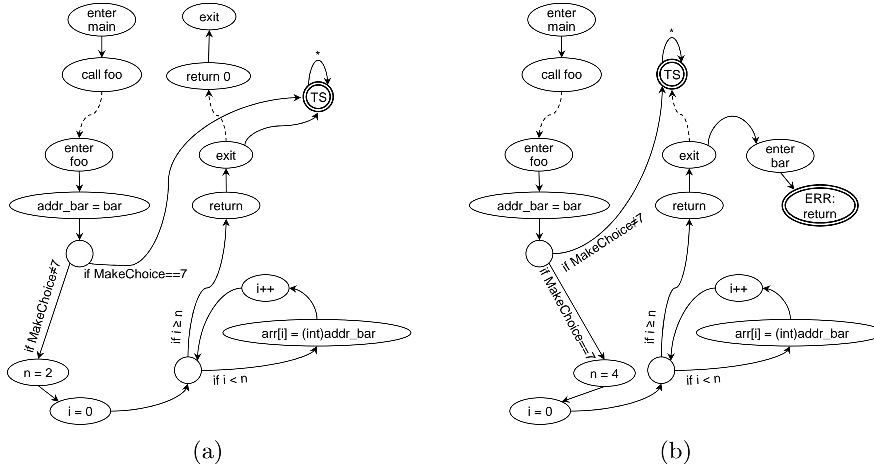

Fig. 4. Automata created by generalizing two execution traces. Each automaton contains an accepting state, called *TS* (for "target surrogate"). *TS* is accepting because it may represent *target*, as well as all non-*target* locations not visited by the trace.

on-the-fly, performing disassembly during state-space exploration, but never on more than one instruction at a time and without relying on a static split between code vs. data. In particular, MCVETO 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 [22]. Second, because the abstraction of the program's state space is built entirely on-the-fly, it allows MCVETO to analyze programs with instruction aliasing. Third, it permits MCVETO to be able to verify (or detect flaws in) self-modifying code. With self-modifying code there is no fixed association between an address and the instruction at that address. However, by labeling each abstract state with a predicate that refers to the address held by the PC, as well as a predicate that specifies what instruction is decoded from memory, starting at the address held by the PC, the abstract graph can capture relationships on an address, the instruction at that address, and the states that can arise for that ⟨address, instruction⟩ combination. A sound overapproximation of the program's state space is created automatically by MCVETO's two mechanisms for refining the abstract graph. Fourth, trace generalization allows eliminating *families* of infeasible traces. Compared to prior techniques that also have this ability [11, 17], the technique involves no calls on an SMT solver, and avoids the potentially expensive step of automaton complementation (see [27, §3.1]).

Returning to the example from Figs. 1 and 2 of a procedure that can corrupt its return address, Fig. 4(a) shows the automaton obtained via trace generalization of the execution trace most likely to be performed during the initial execution. The directed-test-generation step then forces execution down the branch for `MakeChoice()==7`. In that execution, `foo` returns to the beginning of `bar`, from which `ERR` is reachable. (Fig. 4(b) shows the automaton that would be obtained via trace generalization from the second execution trace.)

Fig. 5 shows a program that makes use of instruction aliasing. At line (**), when the instruction is read at the second byte (i.e., starting at `L1+1`), it becomes `L1+1: push 4; call eax`. `ERR` is unreachable because when the branch condition `if(n==1)` is evaluated, `n` always has the value 5: 1 from the initialization in line (*) plus 4 from the value of `eax` added in line (***), which is the return value from the hidden call to `foo` at line (**).

MCVETO builds an abstract graph based on the path

```
n=1; mov eax,0; L1: mov edx,0xd0ff046a; add n,eax; cmp eax,4; jz L2; mov
eax,foo; lea ebx,L1+1; jmp ebx; L1+1: push 4; call eax; return a; add
n,eax; cmp eax,4; jz L2; L2:; if(n==1); return 0
```

It then does a series of refinements of the abstract graph that culminate in a version in which there is no path from the beginning of the graph to `ERR`.

Discovering Candidate Invariants. To improve convergence, we introduced *speculative trace refinement*, which enhances the methods that MCVETO uses to refine the abstract graph. Speculative trace refinement was motivated by the observation that DPG is able to avoid exhaustive loop unrolling if it discovers the right loop invariant. It involves first discovering invariants that hold for nodes of folded traces; the invariants are then incorporated into the abstract graph via automaton intersection. The basic idea is to apply dataflow analysis to a graph obtained from a folded trace to obtain invariants for its states. In the broader context of the full program, these are only *candidate* invariants. They are introduced into the abstract graph in the hope that they are also invariants of the full program. The recovery of invariants is similar in spirit to the computation of invariants from traces in Daikon [13], but in MCVETO they are computed *ex post facto* by dataflow analysis on a folded trace. Although the technique causes the abstract graph to be refined speculatively, the abstract graph is a sound overapproximation of the program's state space at all times.

```
int foo(int a) { return a; }

int main() {
 int n = 1; (*)
 __asm {
     mov eax, 0;
 L1: mov edx, 0xd0ff046a; // (**)
     add n, eax; // (***)
     cmp eax, 4;
     jz L2;
     mov eax, foo;
     lea ebx, L1+1;
     jmp ebx;
 L2: }
 if(n == 1)
     ERR:; // Unreachable
 return 0;
}
```

Fig. 5. A program that illustrates instruction aliasing. At line (**), when the instruction is read at the second byte, it becomes `L1+1: push 4; call eax`.

We take this technique one step further for cases when proxies for program variables are needed in an analysis (e.g., affine-relation analysis [23]). Because no information is available about a program's global and local variables in stripped executables, we perform *aggregate-structure identification* [24] on a concrete trace to obtain a set of inferred memory variables. 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 a folded trace safely, we split n on φ and $\neg\varphi$.

Caveats. MCVETO actually uses nested-word automata [1] rather than finite-state automata to represent the abstract graph and the folded traces that represent generalizations of execution traces. MCVETO makes the assumption that each `call` instruction represents a procedure call, and each `ret` instruction represents a return from a procedure call. This decision was motivated by the desire to have a DPG-based algorithm for verifying machine code that took advantage of the fact that most programs are well-behaved in most execution contexts. The consequence of this decision is that because MCVETO has some expectations on the behaviors of the program, for it to prove that *target* is unreachable it must also prove that the program cannot deviate from the set of expected behaviors (see [27, §3.5]). If a deviant behavior is discovered, it is reported and MCVETO terminates its search.

5 Automatic Tool Generation

Although CodeSurfer/x86 was based on analysis methods that are, in principle, language-independent, the original implementation was tied to the x86 instruction set. That situation is fairly typical of much work on program analysis: although the techniques described in the literature are, in principle, language-independent, implementations are often tied to one specific language. Retargeting them to another language can be an expensive and error-prone process. For machine-code analyses, having a language-dependent implementation is even worse than for source-code analyses because of the size and complexity of instruction sets. Because of instruction-set evolution over time (and the desire to have backward compatibility as word size increased from 8 bits to 64 bits), instruction sets such as the x86 instruction set have several hundred kinds of instructions. Some instruction sets also have special features not found in other instruction sets. To address the problem of supporting multiple instruction sets, another aspect of our work on machine-code analysis has been to develop a meta-tool, or tool-generator, called TSL [21] (for **T**ransformer **S**pecification **L**anguage), to help in the creation of tools for analyzing machine code.

A tool generator (or tool-component generator) such as YACC [18] takes a declarative description of some desired behavior and automatically generates an implementation of a component that behaves in the desired way. Often the generated component consists of generated tables and code, plus some unchanging *driver* code that is used in each generated tool component. The advantage of a tool generator is that it creates correct-by-construction implementations.

For machine-code analysis, the desired components each consist of a suitable abstract interpretation of the instruction set, together with some kind of analysis driver (a solver for finding the fixed-point of a set of dataflow equations, a symbolic evaluator for performing symbolic execution, etc.). TSL is a system that takes a description of the concrete semantics of an instruction set, a description of an abstract interpretation, and creates an implementation of an abstract interpreter for the given instruction set.

TSL : concrete semantics × abstract domain → abstract semantics.

In that sense, TSL is a tool generator that, for a fixed instruction-set semantics, automatically creates different abstract interpreters for the instruction set.

An instruction set's concrete semantics is specified in TSL's input language, which is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching. Writing a TSL specification for an instruction set is similar to writing an interpreter in first-order ML. For instance, the specification of an instruction set's concrete semantics is written as a TSL function

```
state interpInstr(instruction I, state S) {...};
```

where instruction and state are user-defined datatypes that represent the instructions and the semantic states, respectively.

TSL's input language provides a fixed set of base-types; a fixed set of arithmetic, bitwise, relational, and logical operators; and a facility for defining map-types. The meanings of the input-language constructs can be redefined by supplying alternative interpretations of them. When semantic reinterpretation is performed in this way—namely, on the operations of the input-language—it is independent of any given instruction set. Consequently, once a reinterpretation has been defined that reinterprets TSL in a manner appropriate for some state-space-exploration method, the same reinterpretation can be applied to each instruction set whose semantics has been specified in TSL.

The reinterpretation mechanism allows TSL to be used to implement *tool-component generators* and *tool generators*. Each implementation of an analysis component's driver (e.g., fixed-point-finding solver, symbolic executor) serves as the unchanging driver for use in different instantiations of the analysis component for different instruction sets. The TSL language becomes the specification language for retargeting that analysis component for different instruction sets:

analyzer generator = abstract-semantics generator + analysis driver.

For tools like CodeSurfer/x86 and MCVETO, which incorporate multiple analysis components, we thereby obtain YACC-like tool generators for such tools:

concrete semantics of L → Tool/L.

Moreover, because all analysis components are generated from a single specification of the instruction set's concrete semantics, the generated implementations of the analysis components are guaranteed to be mutually consistent (and also to be consistent with an instruction-set emulator that is generated from the same specification of the concrete semantics).

As an example of the kind of leverage that TSL provides, the most recent incarnation of CodeSurfer/x86—a revised version whose analysis components are implemented via TSL—uses eight separate reinterpretations generated from the TSL specification of the x86 instruction set. The x86 version of MCVETO uses three additional reinterpretations [20] generated from the same TSL specification.

Discussion. MCVETO does not model all aspects of a machine-code program. For instance, it does not model timing-related behavior, the hardware caches, the Interrupt Descriptor Table (necessary for modeling interrupt-handler dispatch),

etc. However, the use of TSL allows additional aspects to be added to the concrete operational semantics, independently from MCVETO's DPG algorithms. For example, although our current TSL description of the x86 instruction set does not model the Interrupt Descriptor Table, that is only a shortcoming of the current description and not of MCVETO's DPG algorithms. If the TSL description of the x86 instruction set were augmented to incorporate the Interrupt Descriptor Table in the semantics, the YACC-like tool-generation capabilities would allow easy regeneration of augmented versions of the emulator and symbolic-analysis components used in MCVETO's DPG algorithm.

Moreover, the use of TSL aids the process of augmenting a system like MCVETO with non-standard semantic instrumentation that allows checking for policy violations. For instance, MCVETO currently uses a non-standard instrumented semantics in which the standard instruction-set semantics is augmented with an auxiliary stack [27, §3.5]. Initially, the auxiliary stack is empty; at each `call` instruction, a copy of the return address pushed on the processor stack is also pushed on the auxiliary stack; at each `ret` instruction, the auxiliary stack is checked to make sure that it is non-empty and that the address popped from the processor stack matches the address popped from the auxiliary stack.

6 Related Work

Machine-code analysis has been gaining increased attention, and by now there is a considerable literature on static, dynamic, and symbolic analysis of machine code. It includes such topics as platforms and infrastructure for performing analysis, improved methods to create CFGs, suitable abstract domains for dataflow analysis of machine code, applications in software engineering and program understanding, verification of safety properties, testing (including discovery of security vulnerabilities), malware analysis, type inference, analysis of cache behavior, proof-carrying code, relating source code to the resulting compiled code, and low-level models of the semantics of high-level code. Space limitations preclude a detailed discussion of related work in this paper. An in-depth discussion of work related to CodeSurfer/x86 can be found in [3].

Acknowledgments

We are grateful to our collaborators at Wisconsin—A. Burton, E. Driscoll, M. Elder, and T. Andersen—and at GrammaTech, Inc.—T. Teitelbaum, D. Melski, S. Yong, T. Johnson, D. Gopan, and A. Loginov —for their many contributions to our work on machine-code analysis and verification.

References

1. R. Alur and P. Madhusudan. Adding nesting structure to words. *JACM*, 56, 2009.
2. G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute.* PhD thesis, C.S. Dept., Univ. of Wisconsin, Madison, WI, Aug. 2007. Tech. Rep. 1603.

3. G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *Trans. on Prog. Lang. and Syst.* (To appear.).
4. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.
5. G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis Symp.*, 2006.
6. G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *Verif., Model Checking, and Abs. Interp.*, 2007.
7. G. Balakrishnan and T. Reps. Analyzing stripped device-driver executables. In *Tools and Algs. for the Construct. and Anal. of Syst.*, 2008.
8. 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.
9. G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *VSTTE*, 2007.
10. N. Beckman, A. Nori, S. Rajamani, and R. Simmons. Proofs from tests. In *Int. Symp. on Softw. Testing and Analysis*, 2008.
11. D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Prog. Lang. Design and Impl.*, 2007.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.
13. 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(1–3), 2007.
14. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Prog. Lang. Design and Impl.*, 2005.
15. P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, 2010.
16. B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. SYNERGY: A new algorithm for property checking. In *Found. of Softw. Eng.*, 2006.
17. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.
18. S. Johnson. YACC: Yet another compiler-compiler. Technical Report Comp. Sci. Tech. Rep. 32, Bell Laboratories, 1975.
19. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.
20. J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. In *Spin Workshop*, 2009.
21. J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *Comp. Construct.*, 2008.
22. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS*, 2003.
23. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symp. on Programming*, 2005.
24. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, 1999.
25. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *Part. Eval. and Semantics-Based Prog. Manip.*, 2006.
26. T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *Asian Symp. on Prog. Lang. and Systems*, 2005.
27. A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. In *CAV*, 2010.