TSL: A System for Generating Abstract Interpreters and its Application to Machine-Code Analysis

JUNGHEE LIM
University of Wisconsin
and
THOMAS REPS
University of Wisconsin and GrammaTech, Inc.

This paper describes the design and implementation of a system, called TSL (for "Transformer Specification Language"), that provides a systematic solution to the problem of creating retargetable tools for analyzing machine code. TSL is a tool generator—i.e., a meta-tool—that automatically creates different abstract interpreters for machine-code instruction sets.

The most challenging technical issue that we faced in designing TSL was how to automate the generation of the set of *abstract transformers* for a given abstract interpretaton of a given instruction set. From a description of the *concrete operational semantics* of an instruction set, together with the datatypes and operations that define an abstract domain, TSL automatically creates the set of abstract transformers for the instructions of the instruction set. TSL advances the state of the art in program analysis because it provides two dimensions of parameterizability: (i) a given analysis component can be retargeted to different instruction sets; (ii) multiple analysis components can be created automatically from a single specification of the concrete operational semantics of the language to be analyzed.

TSL is an *abstract-transformer-generator generator*. The paper describes the principles behind TSL, and discusses how one uses TSL to develop different abstract interpreters.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers; model checking*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution; testing tools*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages; macro and assembly languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

Authors' addresses: J. Lim, Computer Sciences Dept., Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53703, junghee@cs.wisc.edu. T. Reps, Computer Sciences Dept., Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53703, and GrammaTech, Inc., 531 Esty St., Ithaca, NY 14850; reps@cs.wisc.edu.

The work was supported in part by NSF under grants CCF-{0524051, 0540955, 0810053, 0904371}; by ONR under grants N00014-{01-1-0708, 01-1-0796, 09-1-0510, 09-1-0776, 10-M-0251, 11-C-0447}; by ARL under grant W911NF-09-1-0413; by AFRL under grants FA8750-05-C-0179, FA8750-06-C-0249, FA9550-09-1-0279 and FA8650-10-C-7088; by DARPA under cooperative agreement HR0011-12-2-0012; by a donation from GrammaTech, Inc.; and by a Symantec Research Labs Graduate Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring companies or agencies.

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

Portions of this work appeared in the 17th Int. Conf. on Compiler Construction [Lim and Reps 2008], the 16th Int. SPIN Workshop [Lim et al. 2009] and a subsequent journal article [Lim et al. 2011], and the 22nd Int. Conf. on Computer Aided Verification [Thakur et al. 2010], as well as in J. Lim's Ph.D. dissertation [Lim 2011].
© 2012 J. Lim and T. Reps

General Terms: Algorithms, Languages, Security, Theory, Verification
Additional Key Words and Phrases: Abstract interpretation, machine-code analysis, dynamic analysis, symbolic analysis, static analysis, dataflow analysis

1. INTRODUCTION

In recent years, methods to analyze machine-code programs have been receiving increased attention. Two of the factors that motivate such work are (i) source code is often unavailable, and (ii) machine code is closer than source code to what is actually executed, which is often important in the context of computer security. While the tools and techniques that have been developed for analyzing machine code are, in principle, language-independent, implementations are often tied to one specific instruction set. Retargeting them to another instruction set can be an expensive and error-prone process.

This paper describes the design and implementation of a system, called TSL (for "**T**ransformer **S**pecification **L**anguage"),[1] that provides a systematic solution to the problem of creating retargetable tools for analyzing machine code. TSL is a tool generator—i.e., a meta-tool—that automatically creates different abstract interpreters for machine-code instruction sets. More precisely, TSL is an *abstract-transformer-generator generator*. The TSL system provides a language in which a user specifies the concrete operational semantics of an instruction set; from a TSL specification, the TSL compiler generates an intermediate representation that allows the meanings of the input-language constructs to be redefined by supplying alternative interpretations of the primitives of the TSL language (i.e., the TSL base-types, map-types, and operations on values of those types). TSL's run-time system supports the use of such generated abstract-transformer generators for dynamic analysis, static analysis, and symbolic execution.

TSL advances the state of the art in program analysis by providing a YACC-like mechanism for creating the key components of machine-code analyzers: from a *description* of the concrete operational semantics of a given instruction set, TSL automatically creates *implementations* of different abstract interpreters for the instruction set.

In designing the TSL system, the most challenging technical issue that we faced was how to automate the generation of the set of *abstract transformers* for a given abstract interpretaton of a given instruction set. There have been a number of past efforts to create generator tools to support abstract interpretation, including MUG2 [Wilhelm 1981], SPARE [Venkatesh 1989; Venkatesh and Fischer 1992], Steffen's work on harnessing model checking for dataflow analysis [Steffen 1991; 1993], Sharlit [Tjiang and Hennessy 1992], Z [Yi and Harrison, III 1993], PAG [Alt and Martin 1995], OPTIMIX [Assmann 2000], TVLA [Lev-Ami and Sagiv 2000; Reps et al. 2010], HOIST [Regehr and Reid 2004], and RHODIUM [Scherpelz et al. 2007]. However, in all but the last three, the user specifies an *abstract semantics*, but not the *concrete semantics* of the language to be analyzed. Moreover, it is the responsibility of the user to establish, outside of the system, the soundness of

[1] TSL is also used as the name of the system's meta-language.

the abstract semantics with respect to the (generally not-written-down) concrete semantics.

In contrast, a major goal of our work was to adhere closely to the credo of abstract interpretation [Cousot and Cousot 1977]:

—specify the concrete semantics

—obtain an abstract semantics as an abstraction of the concrete semantics.

In particular, a specification of the concrete semantics of the language to be analyzed is an explicit artifact that the TSL compiler receives as input. Consequently, TSL differs from most past work that has attempted to automate the creation of abstract interpreters.

A language's concrete semantics is specified in TSL's meta-language. The meta-language is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching. Thus, writing a TSL specification for an language is similar to writing an interpreter for that language in first-order ML.

TSL provides a fixed set of basetypes and operators, as well as map-types with map-access and (applicative) map-update operations. From a TSL specification, the TSL compiler generates a common intermediate representation (CIR) that allows the meanings of the input-language constructs to be redefined by supplying alternative interpretations of the basetypes, map-types, and the operations on them (also known as "*semantic reinterpretation*"). Because all the abstract operations are defined at the *meta-level*, semantic reinterpretation is independent of any given language defined in TSL. Therefore, each implementation of an analysis component's driver serves as the unchanging driver for use in different instantiations of the analysis component to different languages. The TSL language becomes the specification language for retargeting that analysis component for different languages. Thus, to create $M \times N$ analysis components, the TSL system only requires M specifications of the concrete semantics of a language, and N analysis implementations, i.e., $M + N$ inputs to obtain $M \times N$ analysis-component implementations.

Problem Statement. Our work addresses the following fundamental problem in abstract interpretation:

> Given the concrete semantics for a language, how can one systematically create the associated abstract transformers?

In addition to the theory of abstract interpretation itself [Cousot and Cousot 1977], the inspiration for our work is two-fold:

—Prior work on systems that generate analyzers from the concrete semantics of a language: TVLA, HOIST, and RHODIUM.

—Prior work on semantic reinterpretation [Mycroft and Jones 1985; Jones and Mycroft 1986; Nielson 1989; Malmkjær 1993].

The use of semantic reinterpretation in TSL as the basis for generating abstract transformers is what distinguishes our work from TVLA, HOIST, and RHODIUM. Semantic reinterpretation is discussed in more detail in §2.2 and §3.2.

Our work also addresses the *retargeting problem.* The literature on program analysis is vast, and essentially all of the results described in the literature are, in principle, language-independent. However, their implementations are often tied to one specific language. Retargeting them to another language (as well as implementing a new analysis for the same language) can be an expensive and error-prone process. TSL represents one point in the design space of tools to support retargetable program analyzers, namely, a meta-tool—a tool generator—that automatically creates different abstract interpreters for a language.

Contributions. TSL advances the state of the art in program analysis because it provides two dimensions of parameterizability. In particular,

—a given analysis component can be retargeted to different instruction sets. One merely has to write a TSL specification of the concrete semantics of a given instruction set. In this respect, TSL provides a YACC-like mechanism for creating different instantiations of an analysis component for different languages: from a *description* of the concrete operational semantics, TSL automatically creates *implementations* of different analysis components.

—multiple analysis components can be created automatically from a single specification of the concrete operational semantics of the language to be analyzed. For each new analysis component, the analysis designer merely has to provide a reinterpretation of the basetypes, map-types, and operators of the TSL metalanguage.

Other notable aspects of our work include

—*Support for multiple analysis types.* The system supports several analysis types:
  —classical worklist-based value-propagation analyses
  —transformer-composition-based analyses [Cousot and Cousot 1979; Sharir and Pnueli 1981], which are particularly useful for context-sensitive interprocedural analysis, and for relational analyses
  —unification-based analyses for flow-insensitive interprocedural analysis
  —dynamic analyses (including concrete emulation using concrete semantics)
  —symbolic analyses

—*Implemented analyses.* These mechanisms have been instantiated for a number of specific analyses that are useful for analyzing machine code, including value-set analysis [Balakrishnan 2007; Balakrishnan and Reps 2004] (§4.1.1), affine-relation analysis [Müller-Olm and Seidl 2005; Elder et al. 2011] (§4.1.2), def-use analysis (for memory, registers, and flags) (§4.1.4), aggregate structure identification [Ramalingam et al. 1999] (§4.1.3), and generation of symbolic expressions for an instruction's semantics (§4.1.5).

  Using TSL, we also developed a novel way of applying semantic reinterpretation to create symbolic-analysis primitives automatically for symbolic evaluation, preimage computation, and symbolic composition [Lim et al. 2011].

—*Established applicability.* The capabilities of our approach have been demonstrated by writing specifications for IA32 and PowerPC. These are nearly complete specifications of the integer subset of these languages, and include such features as (1) aliasing among 8-, 16-, and 32-bit registers, e.g., al, ah, ax, and

eax (for IA32), (2) endianness, (3) issues arising due to bounded-word-size arithmetic (overflow/underflow, carry/borrow, shifting, rotation, etc.), and (4) setting of condition codes (and their subsequent interpretation at jump instructions). We have also experimented with sufficiently complex features of other machine-code languages (e.g., register windows for Sun SPARC and conditional execution of instructions for ARM) to know that they fit our specification and implementation models.

TSL has been used to recreate the analysis components employed by CodeSurfer/x86 [Balakrishnan et al. 2005], which is a static-analysis framework for analyzing stripped x86 executables. The TSL-generated analysis components include value-set analysis, affine-relation analysis, def-use analysis (for memory, registers, and flags), and aggregate structure identification. From the TSL specification of PowerPC, we also generated the analysis components needed for a PowerPC version, CodeSurfer/ppc32.

In addition, using TSL-generated primitives for symbolic analysis, we developed a machine-code verification tool, called MCVETO [Thakur et al. 2010] (§4.3), and a concolic-execution-based program-exploration tool, called BCE [Lim and Reps 2010] (§4.4).

—*Evaluation of the benefits of the approach.* As discussed in §5, TSL provides benefits from several standpoints, including (i) development time, and (ii) the precision of TSL-generated abstract transformers.

Organization of the Paper. The remainder of the paper is organized as follows. §2 presents an overview of the TSL system, the principles that lie behind it, and the kinds of applications to which it has been applied. §3 describes TSL in more detail, and considered from three perspectives: (i) how to write a TSL specification (from the point of view of instruction-set-specification developers), (ii) how to write domains for (re)interpreting the TSL basetypes and map-types (from the point of view of reinterpretation developers), and (iii) how to use TSL-generated abstract transformers (from the point of view of tool developers). §4 summarizes some of the analyses that have been written using TSL. §5 presents an evaluation of the costs and benefits of the TSL approach. §6 discusses related work. §7 concludes.

2. OVERVIEW OF THE TSL SYSTEM

The goal of TSL is to provide a systematic way of implementing analyzers that work on machine code. TSL has three classes of users: (i) instruction-set-specification (ISS) developers, (ii) reinterpretation developers, and (iii) tool developers. The ISS developers are involved in specifying the semantics of different instruction sets; the reinterpretation developers are involved in defining abstract domains and reinterpretations for the TSL basetypes; the tool developers are involved in extending the analysis framework. The TSL language allows ISS developers to specify the concrete semantics of an instruction set. The TSL run-time system—defined by a collection of C++ classes—allows analysis developers to easily create analyzers that support dynamic analysis, static analysis, and symbolic analysis of executables written in any instruction set for which a TSL semantic specification has been written.

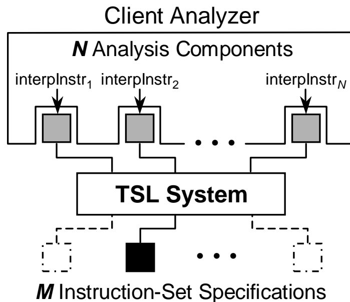

Fig. 1. The interaction between the TSL system and a client analysis tool. Each gray box represents a TSL-generated abstract-transformer generator.

2.1 Design Principles

In designing the TSL language, we were guided by the following principles:

—There should be a formal language for specifying the semantics of the language to be analyzed. Moreover, an ISS developer should specify only the abstract syntax and a concrete operational semantics of the language to be analyzed. Each analyzer should be generated automatically from this specification.

—Concrete syntactic issues—including (i) decoding (machine code to abstract syntax), (ii) encoding (abstract syntax to machine code), (iii) parsing assembly (assembly code to abstract syntax), and (iv) assembly pretty-printing (abstract syntax to assembly code)—should be handled separately from the abstract syntax and concrete semantics.[2]

—There should be a clean interface for reinterpretation developers to specify the abstract semantics for each analysis. An abstract semantics consists of an *interpretation*: an abstract domain and a set of abstract operators (i.e., for the operations of TSL).

—The abstract semantics for each analysis should be separated from the languages to be analyzed so that one does not need to specify multiple versions of an abstract semantics for multiple languages.

Each of these objectives has been achieved in the TSL system: The TSL system translates the TSL specification of each instruction set to a common intermediate representation (CIR) that can be used to create multiple analyzers (§2.3 and §3.1.3). Each analyzer is specified at the level of the meta-language (i.e., by reinterpreting the operations of TSL), which—by extension to TSL expressions and functions—provides the desired reinterpretation of the instructions of an instruction set.

[2]The translation of the concrete syntaxes to and from abstract syntax is handled by a generator tool, called ISAL, which is separate from TSL and will not be discussed in this paper. It suffices to say that the relationship between ISAL and TSL is similar to that between Flex and Bison. With Flex and Bison, the specification of a collection of token identifiers is shared, which allows a Flex-generated lexer to pass tokens to a Bison-generated parser. With ISAL and TSL, the specification of an instruction set's abstract syntax is shared, which allows ISAL to pass abstract-syntax trees to a TSL-generated instruction-set analyzer.

$$s_1\colon x = x \oplus y;$$
$$s_2\colon y = x \oplus y;$$
$$s_3\colon x = x \oplus y;$$

Fig. 2. Code fragment that swaps two `int`s, using three $\oplus$ operations.

Many for the Price of One!. In Fig. 1, once one has the N analysis implementations that are the core of some client analysis tool A, one obtains a generator that can create different versions A/M_1, A/M_2, ... at the cost of writing specifications of the concrete semantics of instruction sets M_1, M_2, etc. Thus, each client analysis tool A built using abstract-transformer generators created via TSL acts as a "YACC-like" tool for generating different versions of A automatically.

2.2 Semantic Reinterpretation

The TSL system is based on factoring the concrete semantics of a language into two parts: (i) a *client* specification, and (ii) a semantic *core*. The interface to the core consists of certain basetypes, map-types, and operators (sometimes called a *semantic algebra* [Schmidt 1986]), and the client is expressed in terms of this interface. This organization permits the core to be *reinterpreted* to produce an alternative semantics for the *subject language*.[3]

Semantic Reinterpretation for Abstract Interpretation. The idea of exploiting such a factoring comes from the field of abstract interpretation [Cousot and Cousot 1977], where factoring-plus-reinterpretation has been proposed as a convenient tool for formulating abstract interpretations and proving them to be sound [Mycroft and Jones 1985; Jones and Mycroft 1986; Nielson 1989; Malmkjær 1993]. In particular, soundness of the *entire* abstract semantics can be established via purely *local* soundness arguments for each of the reinterpreted operators.

The following example shows the basic principles of semantic reinterpretation in the context of abstract interpretation. We use a simple language of assignments, and define the concrete semantics and an abstract sign-analysis semantics via semantic reinterpretation.

Example 2.1. (Adapted from [Malmkjær 1993].) Consider the following fragment of a denotational semantics, which defines the meaning of assignment statements over variables that hold signed 32-bit `int` values (where $\oplus$ denotes exclusive-

[3]Semantic reinterpretation is a program-generation technique, and thus we follow the terminology of the partial-evaluation literature [Jones et al. 1993], where the program on which the partial evaluator operates is called the *subject program*.

In logic and linguistics, the programming language would be called the "object language". In the compiler literature, an object program is a machine-code program produced by a compiler, and so we avoid using the term "object programs" for the programs that TSL operates on.

or):

$$I \in Id \qquad\qquad E \in Expr ::= I \mid E_1 \oplus E_2 \mid \ldots$$
$$S \in Stmt ::= I = E; \qquad\qquad \sigma \in State = Id \rightarrow Int32$$

$$\mathcal{E} : Expr \rightarrow State \rightarrow Int32$$
$$\mathcal{E}[\![I]\!]\sigma = \sigma I$$
$$\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma \oplus \mathcal{E}[\![E_2]\!]\sigma$$

$$\mathcal{I} : Stmt \rightarrow State \rightarrow State$$
$$\mathcal{I}[\![I = E;]\!]\sigma = \sigma[I \mapsto \mathcal{E}[\![E]\!]\sigma]$$

By "$\sigma[I \mapsto v]$," we mean the function that acts like σ except that argument I is mapped to v. The specification given above can be factored into client and core specifications by introducing a domain *Val*, as well as operators *xor*, *lookup*, and *store*. The client specification is defined by

$$xor : Val \rightarrow Val \rightarrow Val$$
$$lookup : State \rightarrow Id \rightarrow Val$$
$$store : State \rightarrow Id \rightarrow Val \rightarrow State$$

$$\mathcal{E} : Expr \rightarrow State \rightarrow Val$$
$$\mathcal{E}[\![I]\!]\sigma = lookup\ \sigma\ I$$
$$\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma\ xor\ \mathcal{E}[\![E_2]\!]\sigma$$

$$\mathcal{I} : Stmt \rightarrow State \rightarrow State$$
$$\mathcal{I}[\![I = E;]\!]\sigma = store\ \sigma\ I\ \mathcal{E}[\![E]\!]\sigma$$

For the concrete (or "standard") semantics, the semantic core is defined by

$$v \in Val_{std} = Int32 \qquad lookup_{std} = \lambda\sigma.\lambda I.\sigma I$$
$$State_{std} = Id \rightarrow Val \qquad store_{std} = \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v]$$
$$xor_{std} = \lambda v_1.\lambda v_2.v_1 \oplus v_2$$

Different abstract interpretations can be defined by using the same client semantics, but giving different interpretations to the basetypes, map-types, and operators of the core. For example, for sign analysis, assuming that *Int32* values are represented in two's-complement notation, the semantic core is reinterpreted as follows:

$$v \in Val_{abs} = \{neg, zero, pos\}^\top$$
$$State_{abs} = Id \rightarrow Val_{abs}$$
$$lookup_{abs} = \lambda\sigma.\lambda I.\sigma I$$
$$store_{abs} = \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v]$$

$xor_{abs} = \lambda v_1.\lambda v_2.$

		v_2: *neg*	*zero*	*pos*	$\top$
v_1	*neg*	$\top$	*neg*	*neg*	$\top$
	zero	*neg*	*zero*	*pos*	$\top$
	pos	*neg*	*pos*	$\top$	$\top$
	$\top$	$\top$	$\top$	$\top$	$\top$

For numbers represented in two's-complement notation, $pos\ xor_{abs}\ neg = neg\ xor_{abs}\ pos = neg$ because, for all combinations of values represented by *pos*

$\sigma_0 := \{x \mapsto neg, y \mapsto pos\}$
$\sigma_1 := \mathcal{I}[\![s_1 : x = x \oplus y;]\!]\sigma_0 = store_{abs}\ \sigma_0\ x\ (neg\ xor_{abs}\ pos) = \{x \mapsto neg, y \mapsto pos\}$
$\sigma_2 := \mathcal{I}[\![s_2 : y = x \oplus y;]\!]\sigma_1 = store_{abs}\ \sigma_1\ y\ (neg\ xor_{abs}\ pos) = \{x \mapsto neg, y \mapsto neg\}$
$\sigma_3 := \mathcal{I}[\![s_3 : x = x \oplus y;]\!]\sigma_2 = store_{abs}\ \sigma_2\ x\ (neg\ xor_{abs}\ neg) = \{x \mapsto \top, y \mapsto neg\}.$

Fig. 3. Application of the abstract transformers created by the sign-analysis reinterpretation to the initial abstract state $\sigma_0 = \{x \mapsto neg, y \mapsto pos\}$.

and *neg*, the high-order bit of the result is set, which means that the result is always negative. However, $pos\ xor_{abs}\ pos = neg\ xor_{abs}\ neg = \top$ because the concrete result could be either 0 or positive, and $zero \sqcup pos = \top$.

For the code fragment shown in Fig. 2, which swaps two `int`s, sign-analysis reinterpretation creates abstract transformers that, given the initial abstract state $\sigma_0 = \{x \mapsto neg, y \mapsto pos\}$, produce the abstract states shown in Fig. 3. □

Alternatives to Semantic Reinterpretation. The mapping of a client specification to the operations of the semantic core resembles a translation to a *universal assembly language* (UAL). Thus, another approach to obtaining "systematic" reinterpretations that are similar to semantic reinterpretations—in that they can be retargeted to multiple subject languages—is to translate subject-language programs to a UAL, and then retarget the instructions of the UAL to operate on abstract values or abstract states. Semantic reinterpretation is compared with this approach in §6.2.

2.3 Technical Contributions Incorporated in the TSL Compilation Process

The specific technical contributions incorporated in the part of the TSL compiler that generates the CIR can be summarized as follows:

—*Two-Level Semantics:* In the TSL system, the notion of a *two-level* intermediate language [Nielson and Nielson 1992] is used to generate the CIR in a way that reduces the loss of precision that could otherwise come about with certain reinterpretations. To address this issue, the TSL compiler performs binding-time analysis [Jones et al. 1993] on the TSL specification to identify which values can always be treated as concrete values, and which operations should therefore be performed in the concrete domain (i.e., should not be reinterpreted). §3.2.2 discusses more details of the two-level intermediate language along with binding-time analysis.

—*Abstract Interpretation:* From a specification, the TSL compiler generates a CIR that has the ability (i) to execute over abstract states, (ii) possibly propagate abstract states to more than one successor in a conditional expression, (iii) compare abstract states and terminate abstract execution when a fixed point is reached, and (iv) apply widening operators, if necessary, to ensure termination. §3.2.1 contains a detailed discussion of these issues.

—*Paired Semantics:* The TSL system allows easy instantiations of *reduced products* by means of *paired semantics*. The CIR can be instantiated with a *paired* semantic domain that couples two interpretations. Communication between the values carried by the two interpretations may take place in the TSL basetype and map-

type operators. §3.2.3 discusses more details of paired semantics.

2.4 The Context of Our Work

TSL has primarily been applied to the creation of abstract interpreters for machine code. Machine-code analysis presents many interesting 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 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.

For more discussion of the challenges and opportunities that arise in machine-code analysis, the reader is referred to [Balakrishnan and Reps 2010] and [Reps et al. 2010]. However, it is worth mentioning a couple of points here that help to illustrate the scope of the problem that TSL addresses:

—The paper presents several fragments of TSL specifications that specify the operational semantics of instruction sets, such as IA32 (also known as x86) and PowerPC (see §4). In such specifications, the subject language is modeled at the level of the instruction-set semantics. That is, the TSL specification describes how the execution of each instruction changes the execution state. Lower-level hardware operations, such as pipelining and paging, are not modeled in our specifications, although TSL is powerful enough to specify such lower-level operations.

—Given a TSL specification of an interpreter at the instruction-set level, the TSL compiler generates an analysis component that performs abstract interpretation on a per-instruction basis. It is the tool developer's responsibility to complete the implementation of the analysis by handling the higher levels of abstract interpretation, such as (i) the propagation of abstract values through the control-flow graph, (ii) determining when a fixed point is reached, etc.

To help out in this task, the TSL system provides several kinds of generic execution/analysis engines that can be instantiated to create finished analyses, including (i) a worklist-based solver for abstract-value propagation over a control-flow graph (for static analysis), (ii) an instruction emulator (for dynamic analysis), and (iii) an engine for performing symbolic execution along a path (for symbolic analysis), as well as a solver for aggregate-structure-identification problems (ASI) [Ramalingam et al. 1999; Balakrishnan and Reps 2007]—a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program.

In addition, we have used TSL-generated abstract transformers with general-purpose analysis packages, such as the WALi system [WALi 2007] for weighted pushdown systems (WPDSs) [Reps et al. 2005; Bouajjani et al. 2003] and the OpenNWA system [Driscoll et al. 2012] for nested-word automata [Alur and

Madhusudan 2006]. In principle, it would be easy to use TSL to drive other similar external packages, such as the Banshee system for solving set-constraint problems [Kodumal and Aiken 2005]. (See §4.1.)

—Although this paper only discusses the application of TSL to machine-code instruction sets, only small extensions would be needed to be able to apply TSL to source-code languages (i.e., to create language-independent analyzers for source-level IRs), as well as bytecode. The main obstacle is that the concrete semantics of a source-code language generally uses an execution state based on a stack of variable-to-value (or variable-to-location, location-to-value) maps. For a machine-code language, the state incorporates an address-based memory model, for which the TSL language provides appropriate primitives.

3. TRANSFORMER SPECIFICATION LANGUAGE

This section presents the basic elements of the TSL system. §3.1 describes the basic elements of the TSL language and what is produced by the TSL compiler. It considers the TSL system from the perspective of instruction-set specifiers (ISS), reinterpretation developers, and tool developers. §3.2 discusses how the TSL compiler generates a CIR from a TSL specification and how the CIR is instantiated for creating analysis components. §3.2 also describes how the TSL system handles some important issues, such as recursion and conditional branches in the CIR. §3.3 discusses the leverage that the TSL system provides.

3.1 Overview of the TSL Language and its Compilation

The key principle of the TSL system is the separation of the semantics of a subject language from the analysis semantics in the development of an analysis component. As discussed in §2.2, the TSL system is based on semantic reinterpretation, which was originally proposed as a convenient *methodology* for formulating abstract interpretations [Cousot and Cousot 1977; Mycroft and Jones 1985; Jones and Mycroft 1986; Malmkjær 1993; Nielson 1989] (see §2.2). Semantic reinterpretation involves refactoring the specification of the concrete semantics of a language into two parts: (i) a *client* specification, and (ii) a semantic *core*. The client is expressed in terms of the semantic core. Such an organization permits the core to be *reinterpreted* to produce an alternative semantics for the subject language.

The key insight behind the TSL system is that if a rich enough *meta-language* is provided for writing semantic specifications, the meta-language itself can serve as the core, and one thereby obtains a suitable client/core factoring for free.

As presented earlier, the TSL system has three classes of users: (i) instruction-set specifiers (ISS), (ii) reinterpretation developers, and (iii) tool developers. The ISS developers use the TSL language to specify the concrete semantics of different instruction sets (the lower part of Fig. 1); the reinterpretation developers use semantic reinterpretation to create new analysis components (the gray boxes in the upper part of Fig. 1).

3.1.1 *TSL from an Instruction-Set Specifier's Standpoint.* Fig. 4 shows part of a specification of the IA32 instruction set taken from the Intel manual [IA32]. The specification describes the syntax and the semantics of each instruction only in a semi-formal way (i.e., a mixture of English and pseudo-code).

General Purpose Registers: EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI,EIP Each of these registers also has 16- or 8-bit subset names. Addressing Modes: [sreg:][offset][([base][,index][,scale])] EFLAGS register: ZF,SF,OF,CF,AF,PF, . . .	ADD r/m32,r32; Add r32 to r/m32 ADD r/m16,r16; Add r16 to r/m16 . . . Operation: DEST ← DEST + SRC; Flags Affected: The OF,SF,ZF,AF,CF, and PF flags are set according to the result.

Fig. 4. A part of the Intel manual's specification of IA32's `add` instruction.

Type	Terms	Constants
BOOL	false, true	false, true
INT64	64-bit signed integers	0d64, 1d64, 2d64, . . .
INT32	32-bit signed integers	0d32, 1d32, 2d32, . . .
INT16	16-bit signed integers	0d16, 1d16, 2d16, . . .
INT8	8-bit signed integers	0d8, 1d8, 2d8, . . .
MAP[α,β]	Maps	$[\alpha \mapsto v_\beta]$

Fig. 5. Basetype and map-type constants. $[\alpha \mapsto v_\beta]$ denotes the map $\lambda x{:}\alpha\,.\,v{:}\beta$.

Our work is based on completely formal specifications that are written in TSL's meta language. TSL is a strongly typed, first-order functional language. TSL supports a fixed set of basetypes; a fixed set of arithmetic, bitwise, relational, and logical operators; the ability to define recursive data-types, map-types, and user-defined functions; and a mechanism for deconstruction by means of pattern matching.

Basetypes. Fig. 5 shows the basetypes that TSL provides. There are two categories of primitive basetypes: *unparameterized* and *parameterized*. An unparameterized basetype is just a set of terms. For example, BOOL is a type consisting of truth values, INT32 is a type consisting of 32-bit signed whole numbers, etc. MAP[α, β] is a predefined parameterized type, with parameters α and β. Each of the following is an instance of the parameterized type MAP:

```
MAP[INT32,INT8]
MAP[INT32,BOOL]
MAP[INT32,MAP[INT8,BOOL]]
```

TSL supports arithmetic/logical operators ($+$, $-$, $*$, $/$, !, &&, ||, xor), bit-manipulation operators ($\sim$, &, |, ˆ, $\ll$, $\gg$, right-rotate, left-rotate), relational operators ($<$, $<=$, $>$, $>=$, $==$, $!=$), and a conditional-expression operator (? :). TSL also provides access/update operators for map-types.

Specifying an Instruction Set. Fig. 6(a) shows a snippet of the TSL specification that corresponds to Fig. 4. (The TSL specification has been pared down to simplify the presentation.)

Much of what an instruction-set specifier writes in a TSL specification is similar to writing an interpreter for an instruction set in first-order ML. One specifies (i) the abstract-syntax grammar of the instruction-set (e.g., lines 2–9 of Fig. 6(a)), (ii) a type for concrete states (e.g., lines 10–12 of Fig. 6(a)), and (iii) the concrete semantics of each instruction (e.g., lines 14–30 of Fig. 6(a)).

Reserved, but User-Defined Types and Reserved Functions. Each specification must define several reserved (but user-defined) types: in Fig. 6(a), instruction (lines 7–9); state—e.g., for 32-bit Intel x86 the type state is a triple of maps (lines 10–

```
[1]  // User-defined abstract syntax
[2]  reg: EAX() | EBX() | . . . ;
[3]  flag: ZF() | SF() | . . . ;
[4]  operand: Indirect(reg reg INT8 INT32)
[5]            | DirectReg(reg)
[6]            | Immediate(INT32) | ...;
[7]   instruction
[8]      : MOV(operand operand)
[9]      | ADD(operand operand| . . . ;
[10] state: State(MAP[INT32,INT8]   // memory-map
[11]               MAP[reg32,INT32]  // register-map
[12]               MAP[flag,BOOL]);   // flag-map
[13] // User-defined functions
[14] INT32 interpOp(state S, operand op) { . . . };
[15] state updateFlag(state S, . . . ) { . . . };
[16] state updateState(state S, . . . ) { . . . };
[17] state interpInstr(instruction I, state S) {
[18]   with(I) (
[19]     MOV(dstOp, srcOp):
[20]       let srcVal = interpOp(S, srcOp);
[21]       in ( updateState( S, dstOp, srcVal ) ),
[22]     ADD(dstOp, srcOp):
[23]       let dstVal = interpOp(S, dstOp);
[24]           srcVal = interpOp(S, srcOp);
[25]           res = dstVal + srcVal;
[26]           S2 = updateFlag(S, dstVal, srcVal, res);
[27]       in ( updateState( S2, dstOp, res ) ),
[28]     . . .
[29]   );
[30] };
```

(a)

```
[1]  template <class INTERP> class CIR {
[2]    class reg { . . . };
[3]    class EAX : public reg { . . . }; . . .
[4]    class flag { . . . };
[5]    class ZF : public flag { . . . }; . . .
[6]    class operand { . . . };
[7]    class Indirect: public operand { . . . }; . . .
[8]    class instruction { . . . };
[9]    class MOV : public instruction { . . .
[10]     operand  op1; operand  op2; . . .
[11]   };
[12]   class ADD : public instruction { . . . }; . . .
[13]   class state { . . . };
[14]   class State: public state { . . . };
[15]   INTERP::INT32 interpOp(state S, operand op) { . . . };
[16]   state updateFlag(state S, . . . ) { . . . };
[17]   state updateState(state S, . . .) { . . . };
[18]   state interpInstr(instruction I, state S) {
[19]    switch(I.id) {
[20]     case ID_MOV: . . .
[21]     case ID_ADD:
[22]      operand dstOp = I.get_child1();
[23]      operand srcOp = I.get_child2();
[24]      INTERP::INT32 dstVal = interpOp(S, dstOp);
[25]      INTERP::INT32 srcVal = interpOp(S, srcOp);
[26]      INTERP::INT32 res = INTERP::Plus(dstVal, srcVal);
[27]      state S2 = updateFlag(S, dstVal, srcVal, res);
[28]      ans = updateState( S2, dstOp, res );
[29]      break;
[30]    . . . }
[31]  }};
```

(b)

Fig. 6. (a) A part of the TSL specification of IA32 concrete semantics, which corresponds to the specification of add from the IA32 manual. Reserved types and function names are underlined, (b) A part of the CIR generated from (a). The CIR is simplified in this presentation.

12); as well as the reserved TSL function interpInstr (lines 17–30). These reserved types and functions form part of the API available to *analysis engines* that use the TSL-generated transformers (i.e., the instantiated CIR).

The definition of types and constructors on lines 2–9 of Fig. 6(a) is an abstract-syntax grammar for IA32. Type reg consists of nullary constructors for the names of the IA32 registers, such as EAX() and EBX(); flag consists of nullary constructors for the names of the IA32 condition codes, such as ZF() and SF(). Lines 4–6 define types and constructors to represent the various kinds of operands that IA32 supports, i.e., various sizes of immediate, direct register, and indirect memory operands. The reserved (but user-defined) type instruction consists of user-defined constructors for each instruction, such as MOV and ADD.

The type state specifies the structure of the execution state. The state for IA32 is defined on lines 10–12 of Fig. 6(a) to consist of three maps, i.e., a memory-map, a register-map, and a flag-map. The *concrete semantics* is specified by writing a function named interpInstr (see lines 17–30 of Fig. 6(a)), which maps an instruction and a state to a state. For instance, the semantics of ADD is to evaluate the two operands in the input state S and create a return state in which the target location holds the summation of the two values and the flags hold appropriate flag values.

3.1.2 *Case Study of Instruction Sets.* In this section, we discuss the quirky characteristics of some instruction sets, and various ways these can be handled in TSL.

IA32. To provide compatibility with 16-bit and 8-bit versions of the instruction set, IA32 provides overlapping register names, such as AX (the lower 16-bits of EAX), AL (the lower 8-bits of AX), and AH (the upper 8-bits of AX). There are two possible ways to specify this feature in TSL. One is to keep three separate maps, for 32-bit registers, 16-bit registers, and 8-bit registers, respectively, and specify that updates to any one of the maps affect the other two maps. Another is to keep one 32-bit map for registers, and obtain the value of a 16-bit or 8-bit register by masking the value of the 32-bit register. (The former can yield more precise VSA results.) Similarly, a 32-bit register is updated with the value of the corresponding 16-bit or 8-bit register by masking and performing a bitwise-or.

The IA32 instruction set keeps condition codes in a special register, called EFLAGS. (Many other instruction sets, such as SPARC, PowerPC, and ARM, also use a special register to store condition codes.) One way to address this feature is to declare "reg32: Eflags();", and make every flag manipulation fetch the bit value from an appropriate bit position of the value associated with Eflags in the register-map. Another way is to introduce flag names, as in our examples, and have every manipulation of EFLAGS affect the entries in a flag-map for the individual flags.

ARM. Almost all ARM instructions contain a condition field that allows an instruction to be executed conditionally, depending on condition-code flags. This feature reduces branch overhead and compensates for the lack of a branch predictor. However, it may worsen the precision of an abstract analysis because in most instructions' specifications, the abstract values from two arms of a TSL conditional expression would be joined.

```
[1] MOVEQ(destReg, srcOprnd):
[2]   let cond = flagMap(EQ());
[3]       src = interpOperand(curState, srcOprnd);
[4]       a = regMap[destReg |-> src];
[5]       b = regMap;
[6]       answer = cond ? a : b;
[7]   in ( answer )
```

Fig. 7. An example of the specification of an ARM conditional-move instruction in TSL.

For example, MOVEQ is one of ARM's conditional instructions; if the flag EQ is true when the instruction starts executing, it executes normally; otherwise, the instruction does nothing. Fig. 7 shows the specification of the instruction in TSL. In many abstract semantics, the conditional expression "*cond* ? *a* : *b*" will be interpreted as a join of the original register map *b* and the updated map *a*, i.e., *join*(*a*,*b*). Consequently, *destReg* would receive the join of its original value and *src*, even when *cond* is known to have a definite value (TRUE or FALSE) in VSA semantics. The paired-semantics mechanism presented in §3.2.3 can help with improving the precision of analyzers by abstractly interpreting conditions. When the

```
[1] reg32 : Reg(INT8) | CWP() | . . .;
[2] reg32 : OutReg(INT8) | InReg(INT8) | . . .;
[3] state: State( . . . , MAP[var32,INT32], . . . );
[4] INT32 RegAccess(MAP[var32,INT32] regmap, reg32 r) {
[5]  let cwp = regmap(CWP());
[6]    key = with(r) (
[7]      OutReg(i):
[8]        Reg(8+i+(16+cwp*16)%(NWINDOWS*16),
[9]      InReg(i): Reg(8+i+cwp*16),
[10]   . . . );
[11] in ( regmap(key) )
[12]}
```

Fig. 8. A method to handle the SPARC register window in TSL.

CIR is instantiated with a paired semantics of VSA_INTERP and DUA_INTERP, and the VSA value of *cond* is FALSE, the DUA_INTERP value for *answer* gets empty *def*- and *use*-sets because the true branch *a* is known to be unreachable according to the VSA_INTERP value of *cond* (instead of non-empty sets for *def*s and *use*s that contain all the definitions and uses in *destReg* and *srcOprnd*).

SPARC. SPARC uses register windows to reduce the overhead associated with saving registers to the stack during a conventional function call. Each window has 8 in, 8 out, 8 local, and 8 global registers. Outs become ins on a context switch, and the new context gets a new set of out and local registers. A specific platform will have some total number of registers, which are organized as a circular buffer; when the buffer becomes full, registers are spilled to the stack to free up a sufficient number for the called procedure. Fig. 8 shows a way to accommodate this feature. The syntactic register (OutReg(n) or InReg(n), defined on line 2) in an instruction is used to obtain a semantic register (Reg(m), defined on line 1, where m represents the register's global index), which is the key used for accesses on and updates to the register map. The desired index of the semantic register is computed from the index of the syntactic register, the value of CWP (the current window pointer) from the current state, and the platform-specific value NWINDOWS (lines 8–9).

3.1.3 *Common Intermediate Representation (CIR).* Fig. 6(b) shows part of the common intermediate representation (CIR) generated by the TSL compiler from Fig. 6(a). (The CIR has been simplified for the presentation in the paper.)

The CIR generated for a given TSL specification is a C++ template that can be used to create multiple analysis components by instantiating the template with different semantic reinterpretations. Each generated CIR is *specific* to a given instruction-set specification, but *common* (whence the name CIR) across generated analyses. Each generated CIR is a template class that takes as input class INTERP, which is an abstract domain for an analysis (line 1 of Fig. 6(b)). The user-defined abstract syntax (lines 2–9 of Fig. 6(a)) is translated to a set of C++ abstract-syntax classes (lines 2–12 of Fig. 6(b)). The user-defined types, such as reg, operand, and instruction, are translated to abstract C++ classes, and the constructors, such as EAX(), Indirect(_,_,_,_), and ADD(_,_), are subclasses of the appropriate parent abstract C++ classes.

Each user-defined function is translated to a CIR function (lines 15–31 of

Fig. 6(b)). Each TSL basetype and basetype-operator is pre-pended with the template parameter name INTERP. To instantiate the CIR, class INTERP is supplied by an analysis developer for the analysis of interest.

The TSL front-end performs *with-normalization*, which transforms all multi-level with expressions to use only one-level patterns, and then compiles the one-level pattern via the pattern-compilation algorithm developed by Wadler [1987] and Pettersson [1992]. Thus, the with expression on line 18 and the patterns on lines 19 and 22 of Fig. 6(a) are translated into switch statements in C++ (lines 19–30 in Fig. 6(b)).

The function calls for obtaining the values of the two operands (lines 23–24 in Fig. 6(a)) correspond to the C++ code on lines 22–25 in Fig. 6(b). The TSL basetype-operator + on line 25 in Fig. 6(a) is translated into a call to INTERP::Plus, as shown on line 26 in Fig. 6(b). The function calls for updating the state (lines 26–27 in Fig. 6(a)) are translated into C++ calls (lines 27–28 in Fig. 6(b)).

§3.2 presents more details about how the CIR is generated and what kind of facilities CIR provides for creating analysis components.

3.1.4 *TSL from a Reinterpretation Developer's Standpoint.* A reinterpretation developer creates a new analysis component by (i) redefining (in C++) the TSL basetypes (BOOL, INT32, INT8, etc.), and (ii) redefining (in C++) the primitive operations on basetypes ($+_{\mathsf{INT32}}$, $+_{\mathsf{INT8}}$, etc.). These are used to instantiate the CIR template by passing a class of basetypes as the template parameter. This approach implicitly defines an alternative interpretation of each expression and function in an instruction-set's concrete semantics (including interpInstr), and thereby yields an alternative semantics for an instruction set from its concrete semantics.

Table I. Parts of the declarations of the basetypes, basetype-operators, and map-access/update functions for three analyses.

VSA	DUA	QFBV
[1] class VSA_INTERP {	[1] class DUA_INTERP {	[1] class QFBV_INTERP {
[2] // basetype	[2] // basetype	[2] // basetype
[3] typedef ValueSet32 INT32;	[3] typedef UseSet INT32;	[3] typedef QFBVTerm32 INT32;
[4] . . .	[4] . . .	[4] . . .
[5] // basetype-operators	[5] // basetype-operators	[5] // basetype-operators
[6] INT32 Plus(INT32 a, INT32 b) {	[6] INT32 Plus(INT32 a, INT32 b) {	[6] INT32 Plus(INT32 a, INT32 b) {
[7] return a.addValueSet(b);	[7] return a.Union(b);	[7] return QFBVPlus32(a, b);
[8] }	[8] }	[8] }
[9] . . .	[9] . . .	[9] . . .
[10] // map-basetypes	[10] // map-basetypes	[10] // map-basetypes
[11] typedef Map<reg32,INT32>	[11] typedef Map<var32,INT32>	[11] typedef QFBVArray
[12] REGMAP32;	[12] REGMAP32;	[12] REGMAP32;
[13] . . .	[13] . . .	[13] . . .
[14] // map-access/update functions	[14] // map-access/update functions	[14] // map-access/update functions
[15] INT32 MapAccess(	[15] INT32 MapAccess(	[15] INT32 MapAccess(
[16] REGMAP32 m, reg32 k) {	[16] REGMAP32 m, reg32 k) {	[16] REGMAP32 m, reg32 k) {
[17] return m.Lookup(k);	[17] return m.Lookup(k);	[17] return QFBVArrayAccess(m,k);
[18] }	[18] }	[18] }
[19] REGMAP32	[19] REGMAP32	[19] REGMAP32
[20] MapUpdate(REGMAP32 m,	[20] MapUpdate(REGMAP32 m,	[20] MapUpdate(REGMAP32 m,
[21] reg32 k, INT32 v) {	[21] reg32 k, INT32 v) {	[21] reg32 k, INT32 v) {
[22] return m.Insert(k, v);	[22] return m.Insert(k,v);	[22] return QFBVArrayUpdate(m,k,v);
[23] }	[23] }	[23] }
[24] . . .	[24]. . .	[24] . . .
[25]};	[25]};	[25]};

Tab. I shows the implementations of primitives for three selected analyses: value-

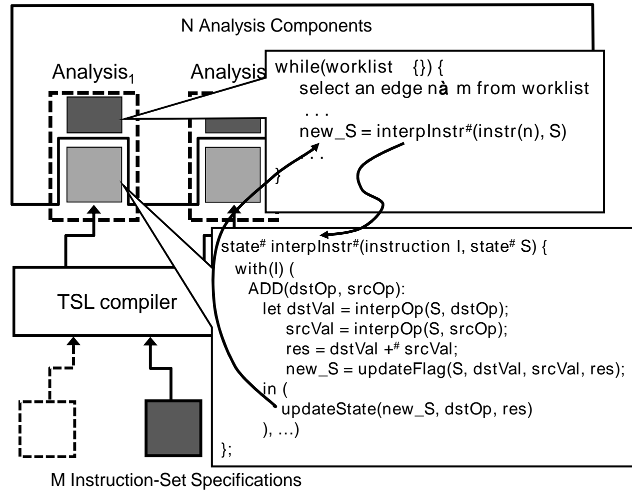

Fig. 9. How a TSL-generated abstract interpreter for instructions, $\text{interpInstr}^\sharp$, is invoked by an analysis engine that performs classical worklist-based propagation of abstract states.

set analysis (VSA, see §4.1.1), def-use analysis (DUA, see §4.1.4), and quantifier-free bit-vector semantics (QFBV, see §4.1.5). Each interpretation defines an abstract domain. For example, line 3 of each column defines the abstract-domain class for INT32: ValueSet32, UseSet, and QFBVTerm32, respectively. To define an interpretation, one needs to define 42 basetype operators, most of which have four variants, for 8-, 16-, 32-, and 64-bit integers, as well as 12 map *access/update* operations. Each abstract domain is also required to contain a set of reserved functions, such as *join*, *meet*, and *widen*, which forms an additional part of the API available to analysis engines that use TSL-generated transformers (see §4).

Usage of TSL-Generated Analysis Components. Fig. 9 shows how the CIR is connected to an analysis engine. The analysis engine in Fig. 9 uses classical worklist-based value propagation in which the TSL-generated transformer interpInstr is invoked with an instruction and the current state S. On each iteration of the main loop of the solver, changes (new_S) would be propagated to successors/predecessors (depending on propagation direction). §4.1 discusses more about how different kinds of analysis engines make use of the CIR.

Generated Transformers. Consider the instruction "add ebx, eax", which causes the sum of the values of the 32-bit registers ebx and eax to be assigned into ebx. When Fig. 6(b) is instantiated with the three interpretations from Tab. I, lines 17–30 of Fig. 6(a) implement the three transformers that are presented (using mathematical notation) in Tab. II.

3.2 More About the Common Intermediate Representation

Given a TSL specification of an instruction set, the TSL system generates a CIR that consists of two parts: one is a list of C++ classes for the user-defined abstract-syntax grammar; the other is a list of C++ template functions for the user-defined

Table II. Semantics of the abstract transformers created using the TSL system.

Analysis	Generated Transformers for "add ebx, eax"
1.VSA	λS.S[ebx $\mapsto$ S(ebx)$+^{vsa}$S(eax)] [ZF $\mapsto$ (S(ebx)$+^{vsa}$S(eax) = 0)][*more flag updates*]
2.DUA	[ebx $\mapsto$ {eax, ebx}, ZF $\mapsto$ {eax, ebx}, . . .]
3.QFBV	(ebx′ = ebx$+^{32}$eax) $\wedge$ (ZF′ $\Leftrightarrow$ (ebx$+^{32}$eax = 0)) $\wedge$ (SF′ $\Leftrightarrow$ (ebx$+^{32}$eax< 0)) $\wedge$ . . .

```
[1]  INTERP::BOOL t0 = . . . ; // translation of a
[2]  INTERP::INT32 t1, t2, answer;
[3]  if(Bool3::possibly_true(t0.getBool3Value())) {
[4]      . . .
[5]      t1 = . . . ; // translation of a
[6]      answer = t1;
[7]  }
[8]  if(Bool3::possibly_false(t0.getBool3Value())) {
[9]      . . .
[10]     t2 = . . . ; // translation of b
[11]     answer = t2;
[12] }
[13] if(t0.getBool3Value() == Bool3::MAYBE) {
[14]     answer = t1.join(t2);
[15] }
```

Fig. 10. The translation of the conditional expression "let answer = cond ? a : b".

functions, including the interface function interpInstr. The C++ functions are generated by linearizing the TSL specification, in evaluation order, into a series of C++ statements, as illustrated by Fig. 6(b). However, there are several issues—discussed below—that need to be properly handled for the resulting code to be suitable for abstract interpretation via semantic reinterpretation.

—§3.2.1 concerns the basic properties needed so that the code can be executed over an abstract domain.

—§3.2.2 discusses a technique that is needed with some generated abstract transformers to side-step a loss of precision during abstract interpretation that would otherwise occur.

—§3.2.3 presents the *paired-semantics* facility that the TSL system provides.

3.2.1 *Execution over an Abstract Domain.* There are four basic properties that the CIR code must support so that it can be executed over an abstract domain. In particular, the code generated for each transformer must be able to

(1) execute over abstract values and abstract states,

(2) possibly propagate abstract values to more than one successor in a conditional expression,

(3) compare abstract states and terminate abstract execution when a fixed point is reached, and

(4) apply widening operators, if necessary, to ensure termination.

Conditional Expressions. Fig. 10 shows part of the CIR that corresponds to the TSL expression "let answer = cond ? a : b". Bool3 is an abstract domain of Booleans

```
[1]  state repMovsd(state S, INT32 count) {
[2]    count == 0
[3]     ? S
[4]     : with(S) (
[5]         State(mem, regs, flags):
[6]           let direction = flags(DF());
[7]               edi = regs(EDI());
[8]               esi = regs(ESI());
[9]               src = MemAccess_32_8_LE_32(mem, esi);
[10]              newRegs = direction
[11]                 ? regs[EDI()|−>edi-4][ESI()|−>esi-4]
[12]                 : regs[EDI()|−>edi+4][ESI()|−>esi+4]
[13]              newMem = MemUpdate_32_8_LE_32(
[14]                 memory, edi, src);
[15]              newS = State(newMem, newRegs, flags);
[16]          in ( repMovsd(newS, count - 1) )
[17]    )
[18]};
```

(a)

```
[1]  state global_S;
[2]  INTERP::INT32 global_count;
[3]  state global_retval;
[4]  state repMovsd(
[5]    lstate S, INTERP::INT32 count) {
[6]     global_S = ⊥;
[7]     global_count = ⊥;
[8]     global_retval = ⊥;
[9]     return repMovsdAux(S, count);
[10]};
[11]state repMovsdAux(
[12]   state S, INTERP::INT32 count) {
[13]     // Widen and test for convergence
[14]     state tmp_S = global_S ∇ (global_S ⊔ S);
[15]     INTERP::INT32 tmp_count =
[16]                global_count ∇ (global_count ⊔ count);
[17]     if(tmp_S ⊑ global_S
[18]       && tmp_count ⊑ global_count) {
[19]       return global_retval;
[20]     }
[21]     S = tmp_S; global_S = tmp_S;
[22]     count = tmp_count; global_count = tmp_count;
[23]
[24]     // translation of the body of repMovsd
[25]     . . .
[26]     state newS = . . . ;
[27]     state t = repMovsdAux(newS, count - 1);
[28]     global_retval = global_retval ⊔ t;
[29]     return global_retval;
[30]};
```

(b)

Fig. 11. (a) A recursive TSL function, (b) The translation of the recursive function from (a). For simplicity, some mathematical notation is used, including $\sqcup$ (join), ∇ (widening), $\sqsubseteq$ (approximation), and $\perp$ (bottom).

(which consists of three values {TRUE, FALSE, MAYBE}, where MAYBE means "may be TRUE or may be FALSE"). The TSL conditional expression is translated into three if-statements (lines 3–7, lines 8–12, and lines 13–15 in Fig. 10). The body of the first if-statement is executed when the Bool3 value for cond is possibly true (i.e., either TRUE or MAYBE). Likewise, the body of the second if-statement is executed when the Bool3 value for cond is possibly false (i.e., either FALSE or MAYBE). The body of the third if-statement is executed when the Bool3 value for cond is MAYBE. Note that in the body of the third if-statement, answer is overwritten with the *join* of t1 and t2 (line 14).

The Bool3 value for the translation of a TSL BOOL-valued value is fetched by getBool3Value, which is one of the TSL interface functions that each interpretation is required to define for the type BOOL. Each analysis developer decides how to handle conditional branches by defining getBool3Value. It is always sound for getBool3Value to be defined as the constant function that always returns MAYBE. For instance, this constant function is useful when Boolean values cannot be expressed in an abstract domain, such as DUA for which the abstract domain for BOOL is a set of *use*s. For an analysis where Bool3 is itself the abstract domain for type BOOL, such as VSA, getBool3Value returns the Bool3 value from evaluating the translation of a so that either an appropriate branch or both branches can be abstractly executed.

Comparison, Termination, and Widening. Recursion is not often used in TSL specifications, but is needed for handling some instructions that involve iteration, such as the IA32 string-manipulation instructions (STOS, LODS, MOVS, etc., with various REP prefixes), and the PowerPC multiple-word load/store instructions (LMW, STMW, etc). For these instructions, the amount of work performed is controlled either by the value of a register, the value of one or more strings, etc. These instructions can be specified in TSL using recursion.[4] For each recursive function, the TSL system generates a function that appropriately compares abstract values and terminates the recursion if abstract values are found to be equal (i.e., the recursion has reached a fixed point). The function is also prepared to apply the widening operator that the reinterpretation developer has specified for the abstract domain in use.

For example, Fig. 11(a) shows the user-defined TSL function that handles "rep movsd", which copies the contents of one area of memory to a second area.[5] The amount of memory to be copied is passed into the function as the argument count. Fig. 11(b) shows its translation in the CIR. A recursive function like repMovsd (Fig. 11(a)) is automatically split by the TSL compiler into two functions, repMovsd (line 4 of Fig. 11(b)) and repMovsdAux (line 11 of Fig. 11(b)). The TSL system initializes appropriate global variables global_S and global_count (lines 6–8) in repMovsd, and then calls repMovsdAux (line 9). At the beginning of repMovsdAux, it generates statements that widen each of the global variables with respect to the arguments, and test whether all of the global variables have reached a fixpoint (lines 13–17). If so, repMovsdAux returns global_retval (line 19). If not, the body of repMovsdAux is analyzed again (lines 24–27). Note that at the translation of each normal return from repMovsdAux (e.g., line 28), the return value is joined into global_retval. The TSL system requires each reinterpretation developer to define the functions *join* and *widen* for the basetypes of the interpretation used in the analysis.

3.2.2 *Two-Level CIR.* The examples given in Fig. 6(b), Fig. 10, and Fig. 11(b), show slightly simplified versions of CIR code. The TSL system actually generates CIR code in which all the basetypes, basetype-operators, and *access/update* functions are appended with one of two predefined namespaces that define a *two-level* interpretation [Jones and Nielson 1995; Nielson and Nielson 1992]: CONCINTERP for concrete interpretation (i.e., interpretation in the concrete semantics), and ABSINTERP for abstract interpretation. Either CONCINTERP or ABSINTERP would replace the occurrences of INTERP in the example CIR shown in Fig. 6(b), Fig. 10, and Fig. 11(b).

The reason for using a two-level CIR is that the specification of an instruction set often contains some manipulations of values that should always be treated as concrete values. For example, an instruction-set specification developer could follow the approach taken in the PowerPC manual [PowerPC32] and specify variants of the conditional branch instruction (BC, BCA, BCL, BCLA) of PowerPC by interpreting some of the fields in the instruction (AA and LK) to determine which of the four variants is being executed (Fig. 12).

Another reason that this issue arises is that most well-designed instruction sets

[4]Currently, TSL supports only linear tail-recursion.

[5]repMovsd is called by interpInstr, which passes in the value of register ecx, and sets ecx to 0 after

```
[1]  // User-defined abstract-syntax grammar
[2]  instruction: . . .
[3]    | BCx(BOOL BOOL INT32 BOOL BOOL)
[4]    | . . . ;
[5]  // User-defined functions
[6]  state interpInstr(instruction I, state S) {
[7]    . . .
[8]    BCx(BO, BI, target, AA, LK):
[9]      let . . .
[10]       cia = RegValue32(S, CIA()); // current address
[11]       new_ia = (AA ? target          // direct: BCA/BCLA
[12]                    : cia + target); // relative: BC/BCL
[13]       lr = RegValue32(S, LR());    // linkage address
[14]       new_lr =
[15]         (LK ? cia + 4 // change the link register: BCL/BCLA
[16]               : lr); // do not change the link register: BC/BCA
[17]     . . .
[18]}
```

Fig. 12. A fragment of the PowerPC specification for interpreting BCx instructions (BC, BCA, BCL, BCLA). For a given instruction, each of BO, BI, target, AA, and LK will have a specific concrete value.

```
[1]  AddSubInstr(op, dstOp, srcOp): // ADD or SUB
[2]    let dstVal = interpOp(S, dstOp);
[3]        srcVal = interpOp(S, srcOp);
[4]        ans = (op == ADD() ? dstVal + srcVal
[5]                           : dstVal - srcVal); // SUB()
[6]    in ( . . . ),
[7]  . . .
```

Fig. 13. An example of factoring in TSL.

have many regularities, and it is convenient to factor the TSL specification to take advantage of these regularities when specifying the semantics. Such factoring leads to shorter specifications, but leads to the introduction of auxiliary functions in which one of the parameters holds a constant value for a *given* instruction. Fig. 13 shows an example of factoring. The IA32 instructions `add` and `sub` both have two operands and can share the code for fetching the values of the two operands. Lines 4–5 are the instruction-specific operations; the equality expression "op == ADD()" on line 4 can be (and should be) interpreted in concrete semantics.

In both cases, the precision of an abstract transformer can sometimes be improved—and is never made worse—by interpreting subexpressions associated with the manipulation of concrete values in concrete semantics. For instance, consider a TSL expression *let* $v = (b ? 1 : 2)$ that occurs in a context in which b is definitely a concrete value; v will get a precise value—either 1 or 2—when b is concretely interpreted. However, if b is not expressible precisely in a given abstract domain, the conditional expression "$(b ? 1 : 2)$" will be evaluated by joining the two branches, and v will not hold a precise value. (It will hold the abstraction of $\{1, 2\}$.)

---

repMovsd returns.

(a)

```
[1] template <typename INTERP1, typename INTERP2>
[2] class PairedSemantics {
[3]   typedef PairedBaseType<INTERP1::INT32, INTERP2::INT32> INT32;
[4]   . . .
[5]   INT32 MemAccess_32_8_LE_32(MEMMAP32_8 mem, INT32 addr) {
[6]     return INT32(INTERP1::MemAccess_32_8_LE_32(mem.GetFirst(), addr.GetFirst()),
[7]                  INTERP2::MemAccess_32_8_LE_32(mem.GetSecond(), addr.GetSecond()));
[8]   }
[9] };
```

(b)

```
[1] typedef PairedSemantics<VSA_INTERP, DUA_INTERP> DUA;
[2] template<> DUA::INT32 DUA::MemAccess_32_8_LE_32(
[3]                               DUA::MEMMAP32_8 mem, DUA::INT32 addr) {
[4]    DUA::INTERP1::MEMMAP32_8 memory1 = mem.GetFirst();
[5]    DUA::INTERP2::MEMMAP32_8 memory2 = mem.GetSecond();
[6]    DUA::INTERP1::INT32 addr1 = addr.GetFirst();
[7]    DUA::INTERP2::INT32 addr2 = addr.GetSecond();
[8]    DUA::INT32 answer = interact(mem1, mem2, addr1, addr2);
[9]    return answer;
[10]}
```

Fig. 14. (a) A part of the template class for paired semantics; (b) an example of C++ explicit template specialization to create a reduced product.

Binding-time analysis. To address the issue, we perform a kind of binding-time analysis [Jones et al. 1993] on the TSL code, the outcome of which is that expressions associated with the manipulation of concrete values in an instruction are annotated with C, and others with A. We then generate the two-level CIR by appending CONCINTERP for C values, and ABSINTERP for A values. The generated CIR is instantiated for an analysis transformer by defining ABSINTERP. The TSL translator supplies a predefined concrete interpretation for CONCINTERP.

The instruction-set-specification developer annotates the top-level user-defined (but reserved) functions, including interpInstr, with binding-time information.

$$\text{EXPORT } {<}A{>} \text{ interpInstr}({<}C{>}, {<}A{>})$$

The first argument of interpInstr, of type instruction, is annotated with $<C>$, which indicates that all data extracted from an instruction is treated as *concrete*. The second argument of interpInstr, of type state, is annotated with $<A>$, which indicates that all data extracted from state is treated as *abstract*. The return type is also annotated as $<A>$.

Binding-time information is propagated through a TSL specification until a fixed point is reached, or an inconsistency is identified.

3.2.3 *Paired Semantics.* Our system allows easy instantiations of *reduced products* [Cousot and Cousot 1979] by means of *paired semantics*. The TSL system provides a template for paired semantics as shown in Fig. 14(a).

The CIR is instantiated with a *paired* semantic domain defined with two interpretations, INTERP1 and INTERP2 (each of which may itself be a paired semantic domain), as shown on line 1 of Fig. 14(b). The communication between interpretations may take place in basetype-operators or *access/update* functions; Fig. 14(b) is an example of the latter. The two components of the paired-semantics values are deconstructed on lines 4–7 of Fig. 14(b), and the individual INTERP1 and INTERP2 components from *both* inputs can be used (as illustrated by the call to *interact* on line 8 of Fig. 14(b)) to create the paired-semantics return value, answer. Such overridings of basetype-operators and *access/update* functions are done by C++

```
[1] with(op) ( . . .
[2]   Indirect32(base, index, scale, disp):
[3]    let  addr = base
[4]            + index * SignExtend8To32(scale)
[5]            + disp;
[6]         m = MemUpdate_32_8_LE_32(
[7]                      mem,addr,v);
[8] . . .)
```

Fig. 15. A fragment of updateState.

explicit specialization of members of class templates (this is specified in C++ by "template<>"; see line 2 of Fig. 14(b)).

We also found this method of CIR instantiation to be useful to perform a form of reduced product when analyses are split into multiple phases, as in a tool like CodeSurfer/x86. CodeSurfer/x86 carries out many analysis phases, and the application of its sequence of basic analysis phases is itself iterated. On each round, CodeSurfer/x86 applies a sequence of analyses: VSA, DUA, and several others. VSA is the primary workhorse, and it is often desirable for the information acquired by VSA to influence the outcomes of other analysis phases by pairing the VSA interpretation with another interpretation.

We can use the paired-semantics mechanism to obtain desired *multi-phase interactions* among our generated analyzers—typically, by pairing the VSA interpretation with another interpretation. For instance, with DUA_INTERP alone, the information required to obtain abstract memory location(s) for addr is lost because the DUA basetype-operators (used for $+$ and $*$ on lines 4–5 of Fig. 15) just return the union of the arguments' *use* sets. With the pairing of VSA_INTERP with DUA_INTERP (line 1 of Fig. 14(b)), DUA can use the abstract address computed for addr2 (line 7 of Fig. 14(b)) by VSA_INTERP, which uses VSA_INTERP::Plus and VSA_INTERP::Times; the latter operators operate on a numeric abstract domain (rather than a set-based one).

Note that during the application of the paired semantics, VSA interpretation will be carried out on the VSA component of paired intermediate values. In some sense, this is duplicated work; however, a paired semantics is typically used only in a phase of transformer generation where the transformers are generated during a single pass over the interprocedural CFG to generate a transformer for each instruction. Thus, only a limited amount of VSA evaluation is performed (equal to what would be performed to check that the VSA solution is a fixed point).

3.3 Leverage

The TSL system provides two dimensions of parameterizability: different instruction sets and different analyses. Each instruction-set specification developer writes an instruction-set semantics, and each reinterpretation developer defines an abstract domain for a desired analysis by giving an interpretation (i.e., the implementations of TSL basetypes, basetype-operators, and *access/update* functions). Given the inputs from these two classes of users, the TSL system automatically generates an analysis component. Note that the work that an analysis developer performs is TSL-specific but *independent* of each language to be analyzed; from the interpretation that defines an analysis, the abstract transformers for that analysis can be

generated automatically for *every* instruction set for which one has a TSL specification. Thus, to create $M \times N$ analysis components, the TSL system only requires M specifications of the concrete semantics of instruction sets, and N analysis implementations (Fig. 1), i.e., $M + N$ inputs to obtain $M \times N$ analysis-component implementations.

The TSL system provides considerable leverage for implementing analysis tools and experimenting with new ones. New analyses are easily implemented because a clean interface is provided for defining an interpretation.

TSL as a Tool Generator. A tool generator (or tool-component generator) such as YACC [Johnson 1975] 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 $\times$ abstract domain $\rightarrow$ 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.

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, which incorporates multiple analysis components, we thereby obtain YACC-like tool generators for such tools:

concrete semantics of L $\rightarrow$ Tool/L.

Consistency. In addition to leverage and thoroughness, for a system like CodeSurfer/x86—which uses multiple analysis phases—automating the process of creating abstract transformers ensures *semantic consistency*; that is, because analysis implementations are generated from a *single* specification of the instruction set's concrete semantics, this guarantees that a *consistent* view of the concrete semantics is adopted by all of the analyses used in the system.

4. APPLICATIONS

The capabilities of the TSL system have been demonstrated by writing specifications for both the IA32 and PowerPC instruction sets, and then automatically cre-

ating a variety of abstract interpreters from each of the specifications—including dynamic-analysis components, static-analysis components, and symbolic-analysis components.

In this section, we present various abstract interpreters generated using TSL, as well as summarize various program-analysis tools developed by using the TSL-generated abstract interpreters.

4.1 TSL-Generated Abstract Interpreters

As illustrated in Fig. 9, a version of the interface function interpInstr is created for each reinterpretation supplied by a reinterpretation designer. At appropriate moments, each analysis engine calls interpInstr with an instruction being processed either (i) to perform one step of abstract interpretation, or (ii) to obtain an abstract transformer. Analysis engines can be categorized as follows:

—*Worklist-based value propagation (or Transformer application)* [TA]. These perform classical worklist-based value propagation in which generated transformers are applied, and changes are propagated to successors/predecessors (depending on propagation direction). Context sensitivity in such analyses is supported by means of the call-string approach [Sharir and Pnueli 1981]. VSA uses this kind of analysis engine (§4.1.1).

—*Transformer composition* [TC]. These generally perform flow-sensitive, context-sensitive interprocedural analysis. ARA (§4.1.2) uses this kind of analysis engine.

—*Unification-based analyses* [UB]. These perform flow-insensitive interprocedural analysis. ASI (§4.1.3) uses this kind of analysis engine.

For each analysis, the CIR is instantiated with an interpretation provided by a reinterpretation developer. This mechanism provides wide flexibility in how one can couple TSL-generated components to an external package. One approach, used with VSA, is that the analysis engine (written in C++) calls interpInstr directly. In this case, the instantiated CIR serves as a *transformer evaluator*: interpInstr is prepared to receive an instruction and an abstract state, and return an abstract state. Another approach, used in both ASI and in TC analyzers, is employed when interfacing to an analysis framework that has its own input language for specifying abstract transformers. In this case, the instantiated CIR serves as an *abstract-transformer generator*: interpInstr is prepared to receive an instruction and an abstract transformer as the state parameter (often the identity function); interpInstr returns an abstract-transformer specification in the analysis component's input language.

The following subsections discuss how the CIR is instantiated for various analyses.

4.1.1 *Creation of a TA Transformer Evaluator for VSA.* VSA is a combined numeric-analysis and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses that each register and memory location holds at each program point [Balakrishnan and Reps 2004]. A *memory region* is an abstract quantity that represents all runtime activation records of a procedure. To represent a set of numeric values and addresses, VSA uses *value-sets*, where a value-set is a map from memory regions to strided intervals. A strided interval consists of a lower bound lb, a stride s, and an upper bound $lb + ks$, and represents the set of numbers $\{lb,\ lb + s,\ lb + 2s,\ ...,\ lb + ks\}$ [Reps et al. 2006].

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a value-set. The abstract domain for BOOL is Bool3 ({TRUE, FALSE, MAYBE}), where MAYBE means "may be FALSE or may be TRUE". The operators on these domains are described in detail in [Reps et al. 2006].

—*The interpretation of map-types and access/update functions.* An abstract value for a memory map (MEMMAP32_8, MEMMAP64_8, etc.) is a dictionary that maps each abstract memory location (i.e., the abstraction of INT32) to a value-set. An abstract value for a register map (REGMAP32, REGMAP64, etc.) is a dictionary that maps each variable (reg32, reg64, etc.) to a value-set. An abstract value for a flag map (FLAGMAP) is a dictionary that maps a flag to a Bool3. The *access/update* functions access or update these dictionaries.

VSA uses this transformer evaluator to create an output abstract state, given an instruction and an input abstract state. For example, row 1 of Tab. II shows the generated VSA transformer for the instruction "add ebx, eax". The VSA evaluator returns a new abstract state in which ebx is updated with the sum of the values of ebx and eax from the input abstract state and the flags are updated appropriately.

4.1.2 *Creation of a TC Transformer Generator for ARA.* An affine relation is a linear-equality constraint between integer-valued variables. ARA finds affine relations that hold in the program, for a given set of variables. This analysis is used to find induction-variable relationships between registers and memory locations; these help in increasing the precision of VSA when interpreting conditional branches [Balakrishnan 2007].

The principle that is used to create a TC transformer generator is as follows: by interpreting the TSL expression that defines the semantics of an individual instruction using an abstract domain in which values represent transformers, each call to interpInstr will residuate a transformer for the instruction. In the case of ARA, the CIR is instantiated so that for each instruction, the generated transformer operates on an abstract domain whose values are sets of matrices that represent affine transformations on registers and memory locations of the state [Müller-Olm and Seidl 2005].

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a set of linear expressions in which variables are either a register or an abstract memory location—the actual representation of the domain is a set of *column*s that consist of an integer constant and an integer coefficient for each variable. This column represents an affine expression over the values that the variables hold at the beginning of the instruction. The basetype operations are defined so that only a set of linear expressions can be generated; any operation that leads to a non-linear expression, such as Times(eax, ebx), returns TOP, which means that no affine relationship is known to hold.

—*The interpretation of map-types and access/update functions.* An abstract value for a map is a set of matrices of size $(N+1) \times (N+1)$, where N is the number of variables. This abstraction, which is able to find all affine relationships in an affine program, was defined by Müller-Olm and Seidl [Müller-Olm and Seidl 2005]. Each *access* function extracts a set of columns associated with the variable it takes as

an argument, from the set of matrices for its map argument. Each *update* function creates a new set of matrices that reflects the affine transformation associated with the update to the variable in question.

For each instruction, the ARA transformer relates linear-equality relationships that hold before the instruction to those that hold after execution of the instruction.

4.1.3 *Creation of a UB Transformer Generator for ASI.* ASI is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program [Balakrishnan and Reps 2007]. For each instruction, the transformer generator generates a set of ASI commands, each of which is either a command to *split* a memory region or a command to *unify* some portions of memory (and/or some registers). At analysis time, a client analyzer typically applies the transformer generator to each of the instructions in the program, and then feeds the resulting set of ASI commands to an ASI solver to refine the memory regions.

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a set of *dataref*s, where a *dataref* is an access on specific bytes of a register or memory. The arithmetic, logical, and bit-vector operations tag *dataref*s as *non-unifiable dataref*s, which means that they will only be used to generate *split*s.

—*The interpretation of map-types and access/update functions.* An abstract value for a map is a set of *split*s and *unification*s. The *access* functions generate a set of *dataref*s associated with a memory location or register. The *update* functions create a set of *unification*s or *split*s according to the *dataref*s of the data argument.

For example, for the instruction "mov [ebx],eax", when ebx holds the abstract address $AR_foo - 12$, where AR_foo is the memory region for the activation records of procedure *foo*, the ASI transformer generator emits one ASI *unification* command "AR_foo[-12:-9] :=: eax[0:3]".

4.1.4 *Def-Use Analysis (DUA).* *Def-Use* analysis finds the relationships between *definitions* (*defs*) and *uses* of state components (registers, flags, and memory-locations) for each instruction.

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a set of *uses* (i.e., abstractions of the map-keys in states, such as registers, flags, and abstract memory locations), and the operators on this domain perform a set union of their arguments' sets.

—*The interpretation of map-types and access/update functions.* An abstract value for a map is a dictionary that maps each *def* to a set of *uses*. Each *access* function returns the set of *uses* associated with the key parameter. Each *update* function $update(D, k, S)$, where D is a dictionary, k is one of the state components, and S is a set of *uses*, returns an updated dictionary $D[k \mapsto (D(k) \cup S)]$ (or $D[k \mapsto S]$ if a strong update is sound).

The DUA results (e.g., row 2 of Tab. II) are used to create transformers for several additional analyses, such as GMOD analysis [Cooper and Kennedy 1988], which is an analysis to find modified variables for each function f (including variables modified

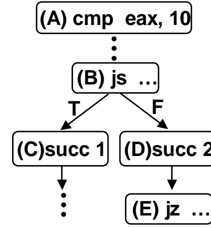

Fig. 16. An example for trace-splitting

by functions transitively called from f) and live-flag analysis, which is used in our version of VSA to perform trace-splitting/collapsing (see §4.1.5).

4.1.5 *Quantifier-Free Bit-Vector (QFBV) Semantics.* QFBV semantics provides a way to obtain a symbolic representation of an instruction's semantics as a formula in first-order quantifier-free bit-vector logic.

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a set of terms, and each operator constructs a term that represents the operation. An abstract value for a BOOL value is a formula, and each BOOL-valued operator constructs a formula that represents the operation.

—*The interpretation of map-types and access/update functions.* An abstract value for a map is a dictionary that maps a storage component to a term (or a formula in the case of FLAGMAP). The *access/update* functions retrieve from and update the dictionaries, respectively.

QFBV semantics is useful for a variety of purposes. One use is as auxiliary information in an abstract interpreter, such as the VSA analysis engine, to provide more precise abstract interpretation of branches in low-level code. The issue is that many instruction sets provide separate instructions for (i) setting flags (based on some condition that is tested) and (ii) branching according to the values held by flags.

To address this problem, we use a *trace-splitting/collapsing* scheme [Mauborgne and Rival 2005]. The VSA analysis engine partitions the state at each flag-setting instruction based on live-flag information (which is obtained from a backwards analysis that uses the DUA transformers). A semantic reduction [Cousot and Cousot 1979] is performed on the split VSA states with respect to a formula obtained from the transformer generated by the QFBV semantics. The set of VSA states that result are propagated to appropriate successors at the branch instruction that uses the flags.

The cmp instruction (A) in Fig. 16, which is a flag-setting instruction, has sf and zf as live flags because those flags are used at the branch instructions js (B) and jz (E): js and jz jump according to sf and zf, respectively. After interpretation of (A), the state S is split into four states, S_1, S_2, S_3, and S_4, which are reduced with respect to the formulas φ_1: (eax $-$ 10 $<$ 0) associated with sf, and φ_2: (eax $-$ 10 $==$ 0) associated with zf.

$S_1 := S[\mathtt{sf}\mapsto T]\ [\mathtt{zf} \mapsto T]\ [\mathsf{eax} \mapsto \mathit{reduce}(S(\mathsf{eax}), \varphi_1 \wedge \varphi_2)]$

$S_2 := S[\mathtt{sf}\mapsto T]\ [\mathtt{zf} \mapsto F]\ [\mathsf{eax} \mapsto \mathit{reduce}(S(\mathsf{eax}), \varphi_1 \wedge \neg\varphi_2)]$

$\mathsf{S}_3 := \mathsf{S}[\mathtt{sf}{\mapsto}\mathsf{F}]\ [\mathtt{zf} \mapsto \mathsf{T}]\ [\mathsf{eax} \mapsto \mathit{reduce}(\mathsf{S}(\mathsf{eax}), \neg\varphi_1 \wedge \varphi_2)]$
$\mathsf{S}_4 := \mathsf{S}[\mathtt{sf}{\mapsto}\mathsf{F}]\ [\mathtt{zf} \mapsto \mathsf{F}]\ [\mathsf{eax} \mapsto \mathit{reduce}(\mathsf{S}(\mathsf{eax}), \neg\varphi_1 \wedge \neg\varphi_2)]$

Because $\varphi_1 \wedge \varphi_2$ is not satisfiable, S_1 becomes $\bot$. State S_2 is propagated to the true branch of js (i.e., just before (C)), and S_3 and S_4 to the false branch (i.e., just before (D)). Because no flags are live just before (C), the splitting mechanism maintains just a single state, and thus all states propagated to (C)—here there is just one—are collapsed to a single abstract state. Because zf is still live until (E), the states S_3 and S_4 are maintained as separate abstract states at (D).

4.2 CodeSurfer/x86

TSL has been used to create a revised version of CodeSurfer/x86 in which the TSL-generated analysis components for VSA, ARA, ASI, QFBV, etc. are put together. CodeSurfer/x86 runs these analyses repeatedly, either until quiescence, or until some user-supplied bound is reached [Balakrishnan and Reps 2010, §6]. It was necessary to find a way in which the abstraction used in each analysis phase could be encoded using TSL's reinterpretation mechanism, and in several cases, including ARA (§4.1.2) and instruction generation for the stand-alone ASI solver (§4.1.3), we were forced to rethink how to implement a particular analysis. In addition, some other analysis techniques were redesigned for the TSL-based version—e.g., an *ad hoc* technique for handling conditional branch instructions during VSA [Balakrishnan and Reps 2010, §3.4.2] was replaced with the trace-splitting method discussed in §4.1.5.

4.3 MCVETO

Using the TSL system, we developed a model checker for machine code, called MCVETO (**M**achine-**C**ode **VE**rification **TO**ol). MCVETO uses *directed proof generation* [Gulavani et al. 2006] 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 on MCVETO 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 does not require static knowledge of the split between code vs. data, and uses a sound approach to disassembly. 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. MCVETO can analyze programs with instruction aliasing[6] because it builds its abstraction of the program's state space

[6]Programs written in instruction sets with varying-length instructions, such as x86, can have "hid-

entirely on-the-fly. Moreover, MCVETO is capable of verifying (or detecting flaws in) self-modifying code. With self-modifying code 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 self-modifying code.

(3) We developed a language-independent algorithm to identify the aliasing condition relevant to a property in a given state. Unlike previous techniques [Beckman et al. 2008], it applies when static names for variables/objects are unavailable.

(4) We developed several techniques to enhance the methods used during directed proof generation to elaborate the abstraction in use.

We were able to develop MCVETO in a language-independent way by using the TSL system to implement the analysis components needed by MCVETO—i.e., (a) an emulator for running tests, (b) a primitive for performing symbolic execution, and (c) a primitive for the pre-image operator. In addition, we developed language-independent approaches to the issues discussed above (e.g., item 3). TSL-generated analysis components, including those for VSA, ARA, and ASI, were used for item 4.

As discussed in §3, the TSL 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 IA32 and PowerPC instruction sets. The details of MCVETO can be found in the full paper ([Thakur et al. 2010]).

4.4 BCE

A substantial number of computers on the Internet have been compromised and have had software installed on them that make them part of a botnet. A typical way to analyze the behavior of a bot executable is to run it and observe its actions. To carry this out, however, one needs to identify inputs that trigger possibly malicious behaviors. Using TSL, we developed a tool for extracting botnet-command information from a bot executable. The tool, called BCE (Botnet-Command Extractor) [Lim and Reps 2010], analyzes a bot executable with the goal of automatically extracting inputs that trigger possibly malicious behaviors.

The contributions of the work on BCE can be summarized as follows:

(1) BCE automates the extraction of information from a bot executable about botnet commands and the arguments to commands. The analysis is carried out without access to source code or symbol-table/debugging information.

(2) BCE is parameterized to take a list of library or system calls of interest ("API calls"). The extracted information includes (a) constant command strings that trigger API-level behaviors; (b) relationships, including type relationships, between the input command string and the actual parameters of an API call; and (c) constraints on the actual parameters of an API call. The information ob-

---

den" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream [Linn and Debray 2003].

tained via BCE can be used to build up input commands that trigger API-level behaviors.

(3) BCE is able to provide a specification of the API-level behaviors of a bot program without running the bot. Along with the input-command strings extracted from a bot program, BCE also provides a sequence of API calls controlled by each command, which can help the user understand the API-level behavior.

(4) BCE uses directed test generation [Godefroid et al. 2005], enhanced with a new search technique that uses control-dependence information [Ferrante et al. 1987] to direct the search. Our experiments showed that the method provides higher coverage of the parts of the program relevant to identifying bot commands, as well as lower overall execution time than standard directed test generation (which does not use control-dependence information).

As with MCVETO, the BCE implementation has been structured so that it can be retargeted to different languages easily. The BCE driver is structured so that one only needs to provide an implementation of concrete execution and symbolic execution of a language. We used the TSL-generated primitives for concrete execution and symbolic execution. The TSL-generated symbolic-analysis primitives allow BCE to obtain accurate path constraints.

The details of BCE can be found in a technical report [Lim and Reps 2010].

5. EVALUATION

In this section, we present an evaluation of the costs and benefits of the TSL approach. The material discussed in this section is designed to shed light on the following questions:

—Does the use of TSL help to reduce the development time of a program-analysis tool that uses abstract interpretation?

—What are the costs in running time and space consumption of using a TSL-generated method for creating abstract transformers?

—In terms of running time, space consumption, and precision, how does the TSL-based method for generating abstract transformers for a given abstract domain $\mathcal{A}$ stack up against a method for creating best abstract transformers [Cousot and Cousot 1979] for $\mathcal{A}$?

In §5.1, we present estimates of the time spent developing two different versions of CodeSurfer/x86.

In §5.2, we report on experiments that compare TSL's approach to creating abstract transformers against best abstract transformers [Cousot and Cousot 1979]: the experiment measures the precision of TSL-generated abstract transformers against the limit of precision obtainable using a given abstraction.

For our experiments at the level of single instructions, we used a corpus of 11,220 x86 instructions, which covers various opcodes, addressing modes, and operand sizes.

Fig. 17 lists several size parameters of a set of Windows utilities (numbers of instructions, procedures, basic blocks, and branches) that we also used in our experiments. For these programs, the generated abstract transformers were used as

Prog. name	Measures of size			
	instrs	procs	BBs	branches
finger	532	18	298	48
subst	1093	16	609	74
label	1167	16	573	103
chkdsk	1468	18	787	119
convert	1927	38	1013	161
route	1982	40	931	243
comp	2377	35	1261	224
logoff	2470	46	1145	306
setup	4751	67	1862	589

Fig. 17. Windows utility applications. The columns show the number of instructions (instrs); the number of procedures (procs); the number of basic blocks (BBs); the number of branch instructions (branches).

"weights" in a weighted pushdown system (WPDS). WPDSs are a modern formalism for solving flow-sensitive, context-sensitive interprocedural dataflow-analysis problems [Reps et al. 2005; Bouajjani et al. 2003].[7] In our experiments, all WPDSs were constructed using the WALi package for WPDSs [WALi 2007]. Weights correspond to abstract transformers for basic blocks.

The asymptotic cost of weight generation is linear in the size of the program: to generate the weights, each basic block in the program is visited once, and a weight is generated by the relevant method: (i) TSL-based reinterpretation of interpInstr to create a weight directly; (ii) TSL-based reinterpretation of interpInstr to create a QFBV formula, followed by application of an algorithm for creating the best weight from the formula (i.e., the best abstract transformer [Cousot and Cousot 1979]).

For each example program, the number of WPDS rules equals the number of basic blocks plus the number of branches (see Fig. 17), together with rules for modeling procedure call and return.

Due to the high cost in §5.2 of constructing WPDSs with best-transformer weights, we ran all WPDS analyses without the code for libraries. Values are returned from x86 procedure calls in register `eax`, and thus in our experiments library functions were modeled approximately (albeit unsoundly, in general) by "`eax := ?`", where "`?`" denotes an unknown value [Müller-Olm and Seidl 2005] (sometimes written as "`havoc(eax)`").

5.1 Development Time

We consider CodeSurfer/x86 to be representative of how TSL can be used to create full-fledged analysis tools; we present our experience developing two different versions of CodeSurfer/x86 [Balakrishnan et al. 2005] as an example of the kind of leverage that TSL provides with respect to development time.

In the original implementation of CodeSurfer/x86, the abstract transformers were implemented in the conventional way (i.e., by hand-writing routines to generate an abstract transformers from an instruction's abstract-syntax tree).[8] The most recent incarnation of CodeSurfer/x86—a revised version whose analysis components

[7]Running a WPDS-based analysis to find the join-over-all-paths value for a given set of program points involves calling two operations, "post*" and "path_summary", as detailed in [Reps et al. 2005].

[8]The ideas used in the abstract transformers, and how the various abstract-interpretation phases

are implemented via TSL—uses eight separate abstract-interpretation phases, each based on a different abstract-transformer generator created from the TSL specification of the IA32 instruction set by supplying an appropriate reinterpretation of the basetypes, map-types, and operations of the TSL meta-language. We estimate that the task of hand-writing an abstract-transformer generator for the eight analysis phases used in the original CodeSurfer/x86 consumed about *twenty man-months*; in contrast, we have invested a total of about *one man-month* to write the C++ code for eight TSL interpretations that are used to generate the replacement components. To this, one should add *10–20 man-days* to write the TSL specification for IA32: the current specification for IA32 consists of 4,153 (non-comment, non-blank) lines of TSL. We conclude that TSL can greatly reduce the time to develop the abstract interpreters needed in a full-fledged analysis tool—perhaps as much as 12-fold (= 20 months/1.67 months).

Because each abstract interpretation is defined at the meta-level (i.e., by providing an interpretation for the collection of TSL primitives), an abstract-transformer generator for a given abstract interpretation can be created automatically for *each* instruction set that is specified in TSL. For instance, from the PowerPC specification (1,862 non-comment, non-blank lines, which took approximately 4 days to write), we were immediately able to generate PowerPC-specific versions of *all* of the abstract-interpretation phases that had been developed for the IA32 instruction set.

It takes approximately 1 minute (on a single core of a four-processor, quad-core, 2.83GHz Intel machine, running Linux 2.6.32 with 8GB of memory, configured so a user process has 4GB of memory) for the TSL (cross-)compiler to compile the IA32 specification to C++. It then takes approximately 4 minutes wall-clock time (on a single core of a four-processor, quad-core, 3GHz Xeon machine with 32GB of RAM, configured so a user process has 4GB of memory, running 64-bit Windows 7 Enterprise with Service Pack 1) to compile the generated C++ for each CIR instantiation, using Visual Studio 2010.

5.2 Comparison to the Best Abstract Transformer

As described in §3, for a given abstract domain $\mathcal{A}$, an over-approximating abstract transformer for an instruction is obtained by defining an over-approximating reinterpretation of each TSL basetype operator as an operation over $\mathcal{A}$. The desired set of abstract transformers are obtained by extending the reinterpretation to TSL expressions and functions, including interpInstr.

However, that method abstracts each TSL operation in isolation, and is therefore rather myopic. In some cases, one can obtain a more precise transformer by considering the semantics of an *entire* instruction (or, even better, an entire basic block or other loop-free path fragment). It is known how to give a *specification* of the most-precise abstract interpretation for a given abstract domain [Cousot and Cousot 1979]: for a Galois connection defined by abstraction function α and concretization function γ, the *best abstract transformer* $\tau^\sharp_{\text{best}}$ for a given concrete transformer τ, defined by

$$\tau^\sharp_{\text{best}} \stackrel{\text{def}}{=} \alpha \circ \tau \circ \gamma, \qquad (1)$$

fit together, are discussed in [Balakrishnan and Reps 2010].

specifies the limit of precision obtainable using a given abstraction.

Unfortunately, Eqn. (1) is non-constructive in general; however, for some abstract domains an algorithm is known for finding best abstract transformers [Graf and Saïdi 1997; Reps et al. 2004; King and Søndergaard 2010; Elder et al. 2011].

To see how a method for constructing best transformers can yield better results than TSL's operator-by-operator reinterpretation approach, consider the following example:

Example 5.1. ([Thakur et al. 2012]) The x86 instruction "`add bh,al`" adds the value of `al`, the low-order byte of 32-bit register `eax`, to `bh`, the second-to-lowest byte of 32-bit register `ebx`. The semantics of this instruction can be expressed in quantifier-free bit-vector (QFBV) logic as follows:

$$\varphi_I \stackrel{\text{def}}{=} \texttt{ebx}' = \begin{pmatrix} (\texttt{ebx} \ \& \ \texttt{0xFFFF00FF}) \\ \mid ((\texttt{ebx} + 256 * (\texttt{eax} \ \& \ \texttt{0xFF})) \ \& \ \texttt{0xFF00}) \end{pmatrix} \wedge \texttt{eax}' = \texttt{eax},$$

where "&" and "|" denote bitwise-and and bitwise-or, respectively. Note that the semantics of the instruction involves non-linear bit-masking operations.

Now suppose that abstract domain is the domain of affine relations over integers mod 2^{32} [Müller-Olm and Seidl 2005; Elder et al. 2011]. For this abstract domain, the best transformer is $(2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx} + 2^{24}\texttt{eax}) \wedge (\texttt{eax}' = \texttt{eax})$, which captures the relationship between the low-order two bytes of `ebx` and the low-order byte of `eax`. It is the best over-approximation to φ_I that can be expressed as an affine relation. In contrast, with TSL's operator-by-operator reinterpretation approach, the abstract transformer obtained from the TSL specification of "`add bh,al`" would be $(\texttt{eax}' = \texttt{eax})$, which loses all information about `ebx`. Such a loss of precision is exacerbated when considering larger loop-free blocks of instructions. □

To compare the TSL operator-by-operator approach with best abstract transformers, we carried out a study using an algorithm for obtaining best abstract transformers for an abstract domain of affine relations. For a given instruction (or basic block) I, the TSL QFBV reinterpretation was used to obtain a formula φ_I that captures the entire semantics of I. The formula φ_I was then used to obtain the best ARA transformer that over-approximates φ_I, using the algorithm described in [Elder et al. 2011; Thakur et al. 2012].

The comparison is based on an abstract domain of affine relations for modular arithmetic, where the variables over which affine relations are inferred are the x86 registers. However, because no algorithm is known for finding best transformers for the affine-relation domain of Müller-Olm and Seidl [2005] (ARA-MOS), we used a related affine-relation domain due to Elder et al. [2011], called ARA-KS, for which such an algorithm is known [Elder et al. 2011].[9] The results reported in this section were obtained using a single core of a four-processor, quad-core, 2.40GHz Xeon machine, running 64-bit Windows XP (Service Pack 2). The machine has 12GB of RAM, configured so a user process has 4GB of memory.

We performed a WPDS-based analysis of the examples from Fig. 17 using three sets of ARA-KS weights (abstract transformers): (i) weights created using the TSL

[9]The domain is called ARA-KS in recognition of the fact that ARA-KS is the extension to modular arithmetic of the abstract domain for Boolean affine relations due to King and Søndergaard [2010]).

reinterpretation method applied to ARA-KS; (ii) weights corresponding to best ARA-KS transformers, and (iii) weights generated using the "generalized Stålmarck" algorithm [Thakur et al. 2012], which provides another point in the design space of trade-offs between time and precision that lies somewhere between (i) and (ii) [Thakur et al. 2012].[10] Note that, except for cases in which an SMT solver timeout is reported, (ii) is guaranteed to find the most-precise basic-block transformers that are expressible in the ARA-KS abstract domain [Elder et al. 2011; Thakur et al. 2012].

5.2.1 *Precision per Instruction.* We compared precision and running time of the "Reinterpretation" method (TSL ARA-KS reinterpretation) versus $\widehat{\alpha}_{\text{KS}}$ at the granularity of individual instructions using the corpus of 11,220 x86 instructions. The experiment showed that the best-transformer method ($\widehat{\alpha}_{\text{KS}}$) is strictly more precise then the ARA-KS Reinterpretation method for only about 2.5% of the instructions—271 out of the 10,964 (= 10,693 + 271) instances of IA32 instructions for which a comparison was possible. The two methods created equal transformers for 10,693 instructions. The best-transformer method uses an SMT solver as a subroutine; with a 1-second timeout limit, there were 256 timeouts in the SMT solver.

In terms of running time, the $\widehat{\alpha}_{\text{KS}}$ method took about 19.5 times longer than the ARA-KS Reinterpretation method.

5.2.2 *Precision in Analysis.* Although the $\widehat{\alpha}_{\text{KS}}$ approach does create more precise abstract transformers than TSL ARA-KS reinterpretation, that only happens for about 2.5% of the instructions. We would like to know what the effect of the increased precision is on the dataflow facts (program invariants) obtained by running an abstract interpreter using the two different sets of abstract transformers. To answer that question, we compared the precision of the ARA-KS results obtained from a flow-sensitive, context-sensitive, interprocedural analysis of the examples from Fig. 17, using ARA-KS abstract transformers generated by the two methods, along with a third method discussed below.

The "Reinterpretation", "Stålmarck", and $\widehat{\alpha}_{\text{KS}}$ methods represent three different points in the design space of trade-offs between time and precision. We performed

[10] To be more precise about the relationship, we used the following "chained" method for generating weights:

(1) "Reinterpretation" is the TSL ARA-KS reinterpretation method.
(2) "Stålmarck" is the generalized-Stålmarck algorithm of Thakur and Reps [2012], *starting with the value obtained via the Reinterpretation method*. The generalized-Stålmarck algorithm successively over-approximates the best transformer from above. By starting the algorithm with the value obtained via the Reinterpretation method, the generalized-Stålmarck algorithm does not have to work its way down from $\top$; it merely continues to work its way down from the over-approximation already obtained via the TSL ARA-KS reinterpretation method. The generalized-Stålmarck algorithm is a faster algorithm than the $\widehat{\alpha}_{\text{KS}}$ method, but does not guarantee to find the best abstract transformer, even in the absence of solver timeouts [Thakur and Reps 2012].
(3) $\widehat{\alpha}_{\text{KS}}$ is the algorithm described in [Elder et al. 2011; Thakur et al. 2012], *starting with the value obtained via the Stålmarck method as an over-approximation* as a way to accelerate its performance. $\widehat{\alpha}_{\text{KS}}$ *does* guarantee to obtain the best abstract transformer, except for cases in which an SMT solver timeout is reported.

Thus, $\widehat{\alpha}_{\text{KS}} \sqsubseteq$ Stålmarck $\sqsubseteq$ Reinterpretation is always guaranteed to hold.

Prog.	Performance (x86)										Precision		
	Reinterp. (RE)			Stålmarck (ST)			$\widehat{\alpha}_{\mathrm{KS}}$				ST < RE	$\widehat{\alpha}_{\mathrm{KS}}$ < ST	$\widehat{\alpha}_{\mathrm{KS}}$ < RE
	WPDS	post*	query	WPDS	post*	query	WPDS	post*	query	t/o			
finger	4.250	0.406	0.110	21.719	0.437	0.110	163.110	0.422	0.125	6	14.6%	10.4%	25.0%
subst	6.203	0.765	0.203	28.484	0.782	0.219	258.094	0.813	0.203	5	12.2%	10.8%	17.6%
label	6.735	0.734	0.282	29.250	0.782	0.297	214.891	0.781	0.297	4	0.1%	0%	0.1%
chkdsk	9.109	0.609	0.312	50.063	0.610	0.297	556.766	0.594	0.313	13	10.9%	0%	10.9%
convert	10.671	1.719	0.453	55.235	1.625	0.422	389.781	1.610	0.422	25	30.4%	0%	30.4%
route	18.969	1.999	0.609	71.109	2.047	0.625	638.016	2.095	0.625	7	22.2%	3.7%	25.1%
comp	14.547	2.016	0.578	67.609	2.141	0.578	770.172	2.140	0.563	8	2.7%	0.5%	3.1%
logoff	23.953	2.625	0.750	98.437	2.751	0.781	805.625	2.655	0.781	28	19.3%	5.9%	23.5%
setup	43.485	1.531	1.329	230.895	1.609	1.250	2061.480	1.624	1.250	86	4.8%	2.2%	5.8%

Fig. 18. WPDS experiments for §5.2. The columns show the times, in seconds, for ARA-KS WPDS construction, performing interprocedural dataflow analysis (running post* and "path_summary"), and finding one-vocabulary affine relations at blocks that end with branch instructions, using three methods: TSL-reinterpretation-based, Stålmarck, and $\widehat{\alpha}_{\mathrm{KS}}$; $\widehat{\alpha}_{\mathrm{KS}}$ has an additional column that reports the number of WPDS rules for which the $\widehat{\alpha}_{\mathrm{KS}}$ weight generation timed out (t/o); the last three columns show the degrees of analysis precision obtained by using (i) Stålmarck-generated ARA-KS weights versus TSL-generated ones, (ii) $\widehat{\alpha}_{\mathrm{KS}}$-generated ARA-KS weights versus Stålmarck-generated ones, and (iii) $\widehat{\alpha}_{\mathrm{KS}}$-generated ARA-KS weights versus TSL-generated ones. (The precision improvements reported in the columns labeled with "A < B" are measured as the percentage of basic blocks that (i) end with a branch instruction, and (ii) begin with a node whose inferred one-vocabulary affine relation via method A was strictly more precise than via method B.)

an experiment designed to illustrate this trade-off. We ran a flow-sensitive, context-sensitive, interprocedural ARA-KS analysis on the corpus of Windows utilities listed in Fig. 17, using the WALi system for weighted pushdown systems (WPDSs), using ARA-MOS abstract transformers generated by the three methods.

For the Windows utilities listed in Fig. 17, Fig. 18 shows the times for constructing ARA-KS abstract transformers, performing interprocedural dataflow analysis (running post* and "path_summary"), and finding one-vocabulary affine relations at blocks that end with branch instructions, using the three methods.

Column 11 of Fig. 18 shows the number of WPDS rules for which $\widehat{\alpha}_{\mathrm{KS}}$ weight-generation timed out. During WPDS construction, if the SMT solver times out, the implementation returns a sound over-approximating weight that the $\widehat{\alpha}_{\mathrm{KS}}$ algorithm has in hand. Because $\widehat{\alpha}_{\mathrm{KS}}$ starts with the weight created by the Stålmarck method, even when there is a timeout, the weight returned by $\widehat{\alpha}_{\mathrm{KS}}$ is never less precise—and sometimes more precise—than the Stålmarck weight. The number of rules is roughly equal to the number of basic blocks plus the number of branches, so a timeout occurred for about 0.6–4.6% of the rules (geometric mean: 1.4%).

The experiment showed that the cost of constructing transformers via the $\widehat{\alpha}_{\mathrm{KS}}$ algorithm is high: creating the ARA-KS weights via $\widehat{\alpha}_{\mathrm{KS}}$ and the Stålmarck-based method are, respectively, about 40.9 times and 4.7 times slower than creating ARA-KS weights using TSL reinterpretation (computed as the geometric mean of the construction-time ratios).

The experiment showed that the TSL ARA-KS reinterpretation method performs quite well in terms of precision, compared to the more precise methods. The precision-improvement numbers show that the $\widehat{\alpha}_{\mathrm{KS}}$ algorithm is strictly more precise than Reinterpretation at about 8.1% of the blocks that end with branch instructions, whereas Stålmarck is strictly more precise than Reinterpretation at about 6.8% of those sites (computed as geometric means).

Prog.	Performance (x86)					
	Reinterp. (RE)		Stålmarck (ST)		$\widehat{\alpha}_{KS}$	
	WPDS	post*	WPDS	post*	WPDS	post*
finger	918	1134	7790	537	3998	540
subst	1729	2080	13242	1176	5177	1176
label	1675	2286	10400	1110	5300	1093
chkdsk	2080	1819	10629	1200	4559	1131
convert	2928	4215	14639	1794	10887	1561
route	3634	5775	13943	2273	9306	2142
comp	3466	8425	13972	2437	8151	1995
logoff	4153	7492	15896	2417	10400	2339
setup	7475	3691	24670	1409	17531	1229

Fig. 19. Space usage in WPDS experiments for §5.2. The columns show the amount of memory used, in KB, for ARA-KS WPDS construction and running post*.

5.2.3 *Memory Requirements.* Fig. 19 shows the space used for the phases of ARA-KS WPDS construction and running post*. The columns of the table in Fig. 19 show the amount of memory used in KB.

The space required for "path_summary" and for finding one-vocabulary affine relations at blocks that end with branch instructions is not reported because in all cases it was negligible.

5.3 Summary and Discussion

Our evaluation of TSL shows that TSL has a number of benefits and relatively few drawbacks, both for affine-relation analysis and the machine-code abstract interpreters used in CodeSurfer/x86.

—TSL can greatly reduce the time to develop the abstract interpreters needed in a full-fledged analysis tool—perhaps as much as 12-fold (§5.1).

—The abstract transformers obtained via a TSL-generated method for creating abstract transformers are nearly as precise as best abstract transformers. As discussed in §5.2.1, the best transformers are more precise for only about 2.5% of the instruction instances in our corpus of 11,220 instructions. Moreover, it takes about 19.5 times longer to obtain best transformers, compared to the TSL-based method.

—When best transformers generated from $\widehat{\alpha}_{KS}$ and Stålmarck are used for interprocedural dataflow analysis, they allow more precise invariants to be discovered at a relatively small number of program points (8.1% and 6.8%, respectively, computed as a geometric mean), although for one example (convert) $\widehat{\alpha}_{KS}$ and Stålmarck obtained more precise invariants at 30.4% of the program points (i.e., at blocks that end with branch instructions).

—Analysis times proper (i.e., the running times in columns "post*" and "query") are not much different when using best transformers, compared to using TSL-based transformers. However, there can be a 40.9-fold performance penalty, with respect to the TSL-based approach, for creating best abstract transformers (i.e., during WPDS construction).

—The Stålmarck-based approach provides another point in the design space of trade-offs between time and precision. The transformers obtained using the

Stålmarck-based approach produce program invariants that are more precise than those found with the TSL-based transformers, but not as precise as those found with the set of best transformers. Moreover, compared to the method for obtaining best abstract transformers, the Stålmarck-based approach has a smaller penalty in running time with respect to the TSL-based approach (Stålmarck is 4.7 times slower).

6. RELATED WORK

In this section, we discuss work from various domains that relates to TSL.

6.1 Instruction-Set Description Languages

There have been many specification languages for instruction sets and many purposes to which they have been applied. Some were designed for hardware simulation, such as cycle simulation and pipeline simulation [Pees et al. 1999; Mishra et al. 2006]. Others have been used to generate an emulator for compiler-optimization testing [Davidson and Fraser 1984; Kästner 2003]. TDL [Kästner 2003] is a hardware-description language that supports the retargeting of back-end phases, such as analyses and optimizations relevant to instruction scheduling, register assignment, and functional-unit binding. The New Jersey machine-code toolkit [Ramsey and Ferandez 1994] addresses concrete syntactic issues (instruction decoding, instruction encoding, etc.).

Siewiorek et al. [1982] proposed an operational hardware specification language, called ISP (Instruction-Set Processor) notation, for describing the instructions in a processor and their semantics. The goal of ISP was to automate the generation of software, the evaluation of computer architectures, and the certification of implementations. They divided a computer system into several levels, including the *program level*, which the ISP notation is designed to describe. The design of the ISP notation was based on two principles:

(1) The components of the program level are a set of memories and a set of operations. The effect of each instruction can be expressed entirely in terms of the information held in the current set of memories. The ISP notation is designed for specifying that a given operation of a processor is performed on a specific data structure that the set of memories hold.

(2) All data operations can be characterized as working on various data-types; each data-type requires distinct operations to process the values of a data-type. A processor can be completely described at the ISP level by giving its instruction set and its interpreter in terms of its operations, data-types, and memories.

TSL relies on the same principles.

While some of the existing instruction-set description *languages* would have been satisfactory for our purposes, their *runtime environments* were not satisfactory, which was what motivated us to implement our own system. In particular, to meet our goals we needed a mechanism to create abstract interpreters of instruction-set specifications. As discussed in §3.2.1, there are four issues that arise. During the abstract interpretation of each instruction, the abstract interpreter must be able to

—execute over abstract values and abstract states,

—execute both branches of a conditional expression,
—compare abstract states and terminate abstract execution when a fixed point is reached, and
—apply widening operators, if necessary, to ensure termination.

As far as we know, TSL is the first instruction-set specification language with support for such mechanisms.

Functional Languages as Instruction-Set Description Language. Harcourt et al. used ML to specify the semantics of instruction sets [Harcourt et al. 1994]. LISAS [Cook et al. 1993] is an instruction-set-description language that was subsequently developed based on their experience using ML. Those two approaches particularly influenced the design of the TSL language.

λ-RTL. TSL shares some of the same goals as λ-RTL [Ramsey and Davidson 1999] (i.e., the ability to specify the semantics of an instruction set and to support multiple clients that make use of a single specification). The two languages were both influenced by ML, but different choices were made about what aspects of ML to retain: λ-RTL is higher-order, but without datatype constructors and recursion; TSL is first-order, but supports both datatype constructors and recursion. Recursion is not often used in specifications, but is needed for handling some loop-iteration instructions, such as the IA32 string-manipulation instructions and the PowerPC multiple-word load/store instructions. The choices made in the design and implementation of TSL were driven by the goal of being able to define multiple abstract interpretations of an instruction-sets semantics.

6.2 Semantic Reinterpretation

As discussed in §2, *semantic reinterpretation* involves refactoring the specification of a language's concrete semantics into a suitable form by introducing appropriate *combinators* that are subsequently redefined to create the different subject-language interpretations.

6.2.1 *Semantic Reinterpretation versus Standard Abstract Interpretation.* Semantic reinterpretation [Mycroft and Jones 1985; Jones and Mycroft 1986; Nielson 1989; Malmkjær 1993] is a form of abstract interpretation [Cousot and Cousot 1977], but differs from the way abstract interpretation is normally applied: in standard abstract interpretation, one reinterprets the constructs of each *subject language*; in contrast, with semantic reinterpretation one reinterprets combinators defined using the *meta-language*. Standard abstract interpretation helps in creating semantically sound *tools*; semantic reinterpretation helps in creating semantically sound *tool generators*. In particular, if you have M subject languages and N analyses, with semantic reinterpretation you obtain $M \times N$ analyzers by writing just $M + N$ specifications: concrete semantics for M subject languages and N reinterpretations. With the standard approach, one must write $M \times N$ abstract semantics.

The MESS compiler generator of Pleban and Lee [1987] aimed to permit the generation of realistic compilers from specifications in denotational semantics. MESS was based on an independent discovery of the principle of semantic reinterpretation: it used a semantic-definition style they called *high-level semantics*, which involves separating the semantic definition of a programming language into two distinct

specifications, called *macro-semantics* and *micro-semantics*. The macro-semantics of a language is defined by a collection of semantic functions that map syntactic phrases compositionally to terms of a semantic algebra; the micro-semantics specifies the meaning of a semantic algebra.

As originally proposed, semantic reinterpretation permits arbitrary refactoring of a semantic specification so that the desired outcome can be achieved via reinterpretation of any combinators introduced. In contrast, although it is possible to introduce combinators in TSL and reinterpret them, the primary mechanism in TSL is to reinterpret the base-types, map-types, and operators of the meta-language. TSL's approach is particularly convenient for a system to generate *multiple* analysis components from a single specification of a language's concrete semantics.

6.2.2 *Semantic Reinterpretation versus Translation to a Universal Assembly Language.* The mapping of subject-language constructs to meta-language operations that one defines as part of the semantic-reinterpretation approach resembles in some ways two other approaches to obtaining "systematic" reinterpretations of subject-language programs, namely,

(1) implementing a translator from subject-language programs to a common intermediate form (CIF) *data structure*, and then creating various interpreters that implement different abstract interpretations of the CIF node types.
(2) implementing a translator from subject-language programs to a universal assembly language (UAL), and then writing different abstract interpreters of the UAL.

An example of a CIF is the Program Intermediate Format of the SUIF compiler infrastructure [Wilson et al. 1994, §2.1]. Recent examples of UALs include Vine [Song et al. 2008, §3], REIL [Dullien and Porst 2009], and BAP [Brumley et al. 2011]. Both approaches can be used to create tools that can be applied to multiple subject languages: each CIF (UAL) abstract interpreter can be applied to the translation of a program written in any subject language L for which one has defined an L-to-CIF (L-to-UAL) translator.

Because UAL programs can be considered to be linearizations of CIF trees, we will confine ourselves to comparing the TSL approach with the UAL approach.

There are four main high-level differences between the semantic-reinterpretation approach used in TSL and the UAL approach. First, there is a difference that affects the activities of instruction-set specifiers: the UAL approach is based on a *translational semantics*, whereas the TSL approach is based on an *operational semantics*. With the UAL approach an instruction-set specifier has to write a function that walks over the abstract-syntax tree of an instruction I of instruction set *IS* and *constructs an appropriate UAL program fragment* whose meaning, when interpreted in the concrete semantics, is the desired semantics for instruction I. In contrast, with TSL one just writes an interpreter that specifies the meaning of an *IS* instruction. (Fig. 6(a), discussed in §3.1.1, shows a fragment of such an interpreter.)

The second and third differences are best explained using Fig. 20.

—The λ expression in Fig. 20(a) formulates the concept underlying the TSL approach. However, in the TSL implementation, the different λ applications are performed in a sequence of phases. Given IS, the definition of a subject lan-

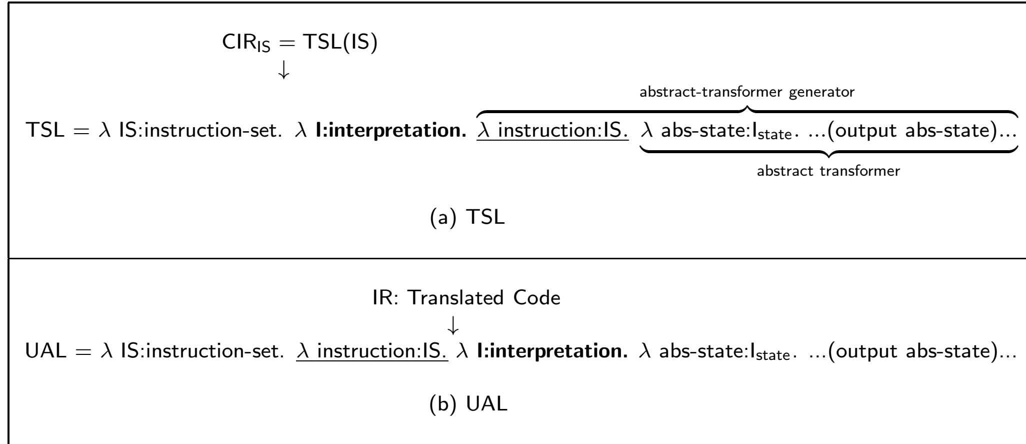

Fig. 20. A comparison of the UAL approach and TSL. (Note the use of dependent-type notation: "instruction:IS" means that instruction is an element of the language IS, and "abs-state:I_{state}" means that abs-state is a state element in the space of values defined by interpretation I.)

guage's syntax, TSL produces an explicit CIR_{IS} artifact as a C++ template. The application of CIR_{IS} to an interpretation I is then performed by the C++ compiler, via template instantiation.

Thus, TSL takes instruction set IS and an interpretation I, and produces $TSL_{IS,I}$ (e.g., $TSL_{x86,VSA}$, $TSL_{x86,QFBV}$, or $TSL_{x86,ARA}$). In some applications, such as ARA, $TSL_{IS,ARA}$ is used as an abstract-transformer generator: given an instruction instr, it produces an ARA abstract-state transformer for instr. In other applications, such as VSA, $TSL_{IS,VSA}$ is used to compute output abstract states; it is given an instruction and an input abstract state and returns an output abstract state.

—The UAL method can be formulated (roughly) as UAL in Fig. 20(b). Conceptually, UAL takes IS, the definition of a subject language's syntax, and instruction, and produces $UAL_{lang,instr}$ (e.g., $UAL_{x86,ADD}$). $UAL_{lang,instr}$ takes an interpretation and an abstract state, and produces an output abstract state.

Thus, the second—and perhaps the most enlightening—difference between the UAL and TSL approaches is the order in which the two parameters I:interpretation and instruction:IS are processed. In the TSL approach, I:interpretation is supplied before instruction:IS; with the UAL approach, they are supplied in the opposite order.

The third difference is due to the fact that, in the implementations of the UAL approach that we are aware of, the parameters IS and instruction:IS are processed first, and an intermediate representation IR consisting of the translated UAL code for instruction:IS is emitted as an explicit data object (see Fig. 20(b)). The final stages—e.g., the processing of interpretation I—involve interpreting a UAL code object. In contrast, with the semantic-reinterpretation approach used in TSL there is no explicit UAL program to be interpreted. In essence, with the semantic-reinterpretation approach as implemented in TSL, a level of interpretation is removed, and hence generated abstract interpreters should run faster.

The fourth difference has to do with the resource costs for the processing carried out by the two methods:

(1) With the UAL method, the size of the IR for a given program P is

$$\sum_{\mathsf{instr} \in P} (\text{size of translation of } \mathsf{instr}) \approx |P| \times (\text{avg. size of translated instruction}).$$

In contrast, the size of the TSL CIR is

$$\#\text{interpretations} \times |\text{OpSem}_L|,$$

where OpSem_L is the operational semantics of language L, and "#interpretations" refers to the number of interpretations in use. In other words, the size of the TSL CIR is constant, independent of the size of the subject-language program, whereas the UAL IR is roughly linear in the size of the subject-language program.

(2) The TSL CIR is a C++ template that is instantiated with a reinterpretation to generate an abstract-transformer generator. Thus, one of the disadvantages of the TSL method is that it is necessary to invoke the C++ compiler to instantiate the CIR. With the UAL method, one only has to recompile the reinterpretation itself, which should generally give faster turnaround time during development.

As discussed in [Lim et al. 2011], although the original target for TSL was the reinterpretation of the semantics of instruction sets, we were able to use TSL to reinterpret the semantics of a *logic*. That is, using the existing facilities of TSL, it was easy to define (i) the abstract syntax of the formulas of a logic, (ii) the logic's standard semantics, and (iii) an abstract interpretation of the logic by reinterpreting TSL's basetypes, map-types, and operators. In this case, we used TSL as the function

$$\mathsf{TSL} = \lambda\ \mathsf{L{:}logic}.\ \lambda\ \mathsf{I{:}semantics}.\ \lambda\ \mathsf{formula{:}L}.\ \lambda\ \mathsf{structure}.\ \mathsf{...(meaning)...}$$

For machine-code analysis, the appropriate logic was quantifier-free bit-vector arithmetic (QFBV; see §4.1.5). Lim et al. [2011] describes the non-standard semantics for logic that we used as the second argument, and how that allowed us to create a pre-image operation for machine-code instruction sets.

Although it is no doubt possible to duplicate what we did using the UAL approach, it would not be straightforward because existing UAL-based systems appear to be less flexible in the languages that can be defined. In contrast, to apply TSL to a logic, we merely had to define a grammar for QFBV abstract-syntax trees, write an interpreter for the standard semantics of QFBV, and define the relevant reinterpretation of TSL's basetypes, map-types, and operators [Lim et al. 2011].

6.3 Systems for Generating Analyzers

Some systems for representing and analyzing programs are (mainly) targeted for a single language. For instance, SOOT [SOOT] is a powerful and flexible analysis/optimization framework that supports analysis and transformation of Java bytecode.

WALA [WALA] is similar to SOOT, but emphasizes a common intermediate form (Common Abstract Syntax Trees), from which multiple additional IRs can be generated (e.g., CFGs and SSA-form). Multiple analyses can then be performed that use these IRs. Several front-ends have been written for WALA, including Java,

Javascript, X10, PHP, Java bytecode, and .NET bytecode. LLVM [Lattner and Adve 2004] is another multi-lingual framework that is similar to WALA. LLVM support a different common intermediate form (LLVM intermediate representation). Languages supported by LLVM include Ada, C, C++, D, Fortran, and Objective-C.

One method to support the retargeting of analyses to different languages is to create a package that supports a family of program analyses that different front-ends can use to create analysis components. Examples include BDDBDDB [Whaley et al. 2005], Banshee [Kodumal and Aiken 2005], APRON [APRON], WPDS++ [WPDS++ 2004], and WALi [WALi 2007]. The writer of each client front-end needs to encode the semantics of his language by creating appropriate transformers for each statement and condition in the subject language's intermediate representation, using the package's API (or input language).

To use such packages in conjunction with TSL, a reinterpretation designer need only use the package to implement the appropriate abstract operations so as to over-approximate the semantics of the operations of the TSL meta-language. Any of the aforementioned packages could be used for creating TSL-based analyses; to date, WALi [WALi 2007] has been used for all of the TC-style analyzers (§4.1.2) that have been developed for use with TSL.

As mentioned in §1, there have been a number of past efforts to create generator tools that support abstract interpretation, including MUG2 [Wilhelm 1981], SPARE [Venkatesh 1989; Venkatesh and Fischer 1992], Steffen's work on harnessing model checking for dataflow analysis [Steffen 1991; 1993], Sharlit [Tjiang and Hennessy 1992], Z [Yi and Harrison, III 1993], PAG [Alt and Martin 1995], and OPTIMIX [Assmann 2000]. In contrast with TSL, in those systems the user specifies the *abstract* semantics of the language to be analyzed, rather than the *concrete* semantics.

Steffen [1991] [Steffen 1993] uses CTL as a specification language for dataflow-analysis algorithms. An advantage of this approach is that it is *declarative*: a specification concerns the program property under consideration, rather than specific details of the analysis algorithm or information about how the properties of interest are determined. However, Steffen's approach does not start from the concrete operational semantics of the language; instead, it starts from an abstraction of the program, which consists of a labeled transition system annotated with a set of atomic propositions. The set of atomic propositions defines the abstract domain in use. Later developments stemming from Steffen's approach include work by Schmidt [1998], Cousot and Cousot [2000], and Lacey et al. [2004].

There are two analysis systems, TVLA [Lev-Ami and Sagiv 2000; Reps et al. 2010] and RHODIUM [Scherpelz et al. 2007], in which sound analysis transformers are generated automatically from a concrete operational semantics, plus a specification of an abstraction (either via the abstraction function (TVLA) or the concretization function (RHODIUM)).

—RHODIUM uses a heuristic method for creating sound abstract transformers for parameterized predicate abstraction [Cousot 2003]. Suppose that the meaning of dataflow fact x, when expressed in logic, is some formula $\hat{\gamma}(x)$, and that the goal is to create the abstract transformer for statement S. The RHODIUM method involves two steps:

—Create the formula $\varphi_x = \text{WLP}(S, \hat{\gamma}(x))$, where WLP is the weakest-liberal-precondition operator.

—Find a Boolean combination $\psi[\hat{\gamma}(D)]$ of pre-state dataflow facts D that under-approximates φ_x. That is, if $\psi[\hat{\gamma}(D)]$ holds in the pre-state, then φ_x must also hold in the pre-state, and hence $\hat{\gamma}(x)$ must hold in the post-state.

The abstract transformer is a function that sets the value of x in the post-state according to whether $\psi[\hat{\gamma}(D)]$ holds in the pre-state.

—The method for automatically generating abstract transformers used in TVLA [Reps et al. 2010] is based on finite-differencing of formulas. In TVLA, the abstraction in use is defined by a set of so-called "instrumentation predicates". An instrumentation predicate captures a property that may be possessed by some components of a state and thus distinguishes them from other components of the state. In other words, the set of instrumentation predicates characterizes the distinctions among elements that are observable in the abstraction.

The problem addressed by Reps et al. [2010] is how to establish the values of post-state instrumentation predicates after the execution of a statement S. Instrumentation predicates are defined by formulas, so the defining formula for a post-state instrumentation predicate p could always be evaluated in the post-state. However, TVLA is based on three-valued logic, in which a third truth value, denoted by 1/2, is introduced to indicate uncertainty (or the absence of information). Evaluating p's defining formula in the post-state often results in 1/2 values, even for values that were "definite" in the pre-state—i.e., either true (1) or false (0)—and could not have been affected by S.

To overcome such loss of precision, TVLA uses an incremental-updating approach: it copies the pre-state values of an instrumentation predicate to the post-state, and only updates entries that are in the "footprint" of S (which is generally small). In essence, the role of finite differencing in the TVLA approach is to identify the effect of S on S's footprint.

HOIST [Regehr and Reid 2004] also generates abstract transformers from the concrete semantics of a machine-code language. However, the construction of an abstract transformer by HOIST is not done by processing the specification symbolically, as in TVLA and RHODIUM. Instead, HOIST accesses the CPU—either a physical CPU or an emulated one—and runs the instruction on specially selected inputs; from the results of these tests, an abstract transformer is obtained—first in the form of a BDD and then as C code. The paper on HOIST reports that it is "limited to eight-bit machines due to costs exponential in the word size of the target architecture".

The use of semantic reinterpretation in TSL as the basis for generating abstract transformers is what distinguishes our work from TVLA, RHODIUM, and HOIST. In TSL, we rely on the analysis developer to supply sound reinterpretations of the basetypes, map-types, and operators of the TSL meta-language. While this requirement places an additional burden on developers, once an analysis is developed it can be used with each instruction set specified in TSL. Moreover,

—the analyses that we support are much more efficient than those that can be created with TVLA and apply to our intended domain of application (abstract interpretation of machine code).

—some of the analyses that we use, such as ARA-MOS [Müller-Olm and Seidl 2005] and ARA-KS [Elder et al. 2011], appear to be beyond the power of the transformer-generation methods developed for use in TVLA, RHODIUM, and HOIST.

7. CONCLUSION

Although essentially all program-analysis techniques described in the literature are language-independent, analysis implementations are often tied to a particular language-specific compiler infrastructure. Unlike the situation in source-code analysis, which can be addressed by developing relatively small common intermediate representations, machine-code analysis suffers from the fact that instruction sets typically have hundreds of instructions and a variety of architecture-specific features that are incompatible with other architectures. With future computing platforms based on multi-core architectures and transactional memory, future run-time environments using just-in-time compiling, future systems providing cloud computing and autonomic computing, plus cell phones, PDAs, wearable computers, and autonomous vehicles all entering the fray, both (i) interest in establishing that security and reliability properties hold for machine code, and (ii) the variety of computing platforms to analyze will only increase.

To help address these concerns, we have developed improved infrastructure for analyzing machine code. Our work is embodied in the TSL language—a language for describing the semantics of an instruction set—as well as in the TSL run-time system, which supports the creation of a multiplicity of static-analysis, dynamic-analysis, and symbolic-analysis components.

Using TSL, we developed several applications for analyzing machine-code, including a revised version of CodeSurfer/x86 in which all analysis components are generated from a TSL specification of the IA32 instruction set. The analogous components for a CodeSurfer/ppc32 system were generated from a TSL specification of the PowerPC32 instruction set.

In addition, we were able to create mutually-consistent, correct-by-construction implementations of symbolic primitives—in particular, quantifier-free, first-order-logic formulas for (i) symbolic evaluation of a single command, (ii) pre-image with respect to a single command, and (iii) symbolic composition for a class of formulas that express state transformations. Using the symbolic-analysis primitives, together with other TSL-generated abstract-interpretation components, we developed two analysis tools—MCVETO and BCE—that use logic-based search procedures to establish properties of machine-code programs.

Our evaluation of TSL in §5 shows that TSL has a number of benefits and relatively few drawbacks.

Acknowledgments. We are grateful to our collaborators currently or formerly at Wisconsin—T. Andersen, G. Balakrishnan, E. Driscoll, M. Elder, N. Kidd, A. Lal, T. Sharma, and A. Thakur—and at GrammaTech, Inc.—T. Teitelbaum, S. Yong, D. Melski, T. Johnson, D. Gopan, and A. Loginov—for their many contributions to the project.

REFERENCES

Alt, M. and Martin, F. 1995. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symp.*

Alur, R. and Madhusudan, P. 2006. Adding nesting structure to words. In *Developments in Lang. Theory*.

APRON. APRON numerical abstract domain library. apron.cri.ensmp.fr.

Assmann, U. 2000. Graph rewrite systems for program optimization. *Trans. on Prog. Lang. and Syst. 22,* 4.

Balakrishnan, G. 2007. WYSINWYX: What You See Is Not What You eXecute. Ph.D. thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1603.

Balakrishnan, G., Gruian, R., Reps, T., and Teitelbaum, T. 2005. Codesurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *Comp. Construct.*

Balakrishnan, G. and Reps, T. 2004. Analyzing memory accesses in x86 executables. In *Comp. Construct.* 5–23.

Balakrishnan, G. and Reps, T. 2007. DIVINE: DIscovering Variables IN Executables. In *Verif., Model Checking, and Abs. Interp.*

Balakrishnan, G. and Reps, T. 2010. WYSINWYX: What You See Is Not What You eXecute. *Trans. on Prog. Lang. and Syst. 32,* 6.

Beckman, N., Nori, A., Rajamani, S., and Simmons, R. 2008. Proofs from tests. In *Int. Symp. on Softw. Testing and Analysis*.

Bouajjani, A., Esparza, J., and Touili, T. 2003. A generic approach to the static analysis of concurrent programs with procedures. In *Princ. of Prog. Lang.* 62–73.

Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. 2011. BAP: A binary analysis platform. In *Computer Aided Verif.*

Cook, T. A., Franzon, P. D., Harcourt, E. A., and Miller, T. K. 1993. System-level specification of instruction sets. In *DAC*.

Cooper, K. and Kennedy, K. 1988. Interprocedural side-effect analysis in linear time. In *Prog. Lang. Design and Impl.* 57–66.

Cousot, P. 2003. Verification by abstract interpretation. In *Verification: Theory and Practice*.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Princ. of Prog. Lang.* 238–252.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *POPL*.

Cousot, P. and Cousot, R. 2000. Temporal abstract interpretation. In *Princ. of Prog. Lang.* 12–25.

Davidson, J. W. and Fraser, C. W. 1984. Code selection through object code optimization. In *TPLS*.

Driscoll, E., Thakur, A., and Reps, T. 2012. OpenNWA: A nested-word-automaton library (tool paper). In *Computer Aided Verif.*

Dullien, T. and Porst, S. 2009. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest*.

Elder, M., Lim, J., Sharma, T., Andersen, T., and Reps, T. 2011. Abstract domains of affine relations. In *Static Analysis Symp.*

Ferrante, J., Ottenstein, K., and Warren, J. 1987. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst. 3,* 9, 319–349.

Godefroid, P., Klarlund, N., and Sen, K. 2005. DART: Directed automated random testing. In *Prog. Lang. Design and Impl.*

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *Computer Aided Verif.* Lec. Notes in Comp. Sci., vol. 1254. 72–83.

Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., and Rajamani, S. 2006. SYNERGY: A new algorithm for property checking. In *Found. of Softw. Eng.*

Harcourt, E., Mauney, J., and Cook, T. 1994. Functional specification and simulation of instruction set architectures. In *PLC*.

IA32. *IA-32 Intel Architecture Software Developer's Manual*. developer.intel.com/design/pentiumii/manuals/243191.htm.

Johnson, S. 1975. YACC: Yet another compiler-compiler. Tech. Rep. Comp. Sci. Tech. Rep. 32, Bell Laboratories.

Jones, N., Gomard, C., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International.

Jones, N. and Mycroft, A. 1986. Data flow analysis of applicative programs using minimal function graphs. In *Princ. of Prog. Lang.* 296–306.

Jones, N. and Nielson, F. 1995. Abstract interpretation: A semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Vol. 4. Oxford Univ. Press, 527–636.

Kästner, D. 2003. TDL: a hardware description language for retargetable postpass optimizations and analyses. In *GPCE*.

King, A. and Søndergaard, H. 2010. Automatic abstraction for congruences. In *Verif., Model Checking, and Abs. Interp.*

Kodumal, J. and Aiken, A. 2005. Banshee: A scalable constraint-based analysis toolkit. In *Static Analysis Symp.*

Lacey, D., Jones, N., Van Wyk, E., and Frederiksen, C. 2004. Compiler optimization correctness by temporal logic. *Higher-Order and Symbolic Computation 17,* 3.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Int. Symp. on Code Generation and Optimization*.

Lev-Ami, T. and Sagiv, M. 2000. TVLA: A system for implementing static analyses. In *Static Analysis Symp.* 280–301.

Lim, J. 2011. Transformer Specification Language: A system for generating analyzers and its applications. Ph.D. thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1689.

Lim, J., Lal, A., and Reps, T. 2009. Symbolic analysis via semantic reinterpretation. In *Spin Workshop*.

Lim, J., Lal, A., and Reps, T. 2011. Symbolic analysis via semantic reinterpretation. *Softw. Tools for Tech. Transfer 13,* 1, 61–87.

Lim, J. and Reps, T. 2008. A system for generating static analyzers for machine instructions. In *Comp. Construct.*

Lim, J. and Reps, T. 2010. BCE: Extracting botnet commands from bot executables. Tech. Rep. TR-1668.

Linn, C. and Debray, S. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *CCCS*.

Malmkjær, K. 1993. Abstract interpretation of partial-evaluation algorithms. Ph.D. thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, Kansas.

Mauborgne, L. and Rival, X. 2005. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*.

Mishra, P., Shrivastava, A., and Dutt, N. 2006. Architecture description language: driven software toolkit generation for architectural exploration of programmable SOCs. *TODAES*.

Müller-Olm, M. and Seidl, H. 2005. Analysis of modular arithmetic. In *European Symp. on Programming*.

Mycroft, A. and Jones, N. 1985. A relational framework for abstract interpretation. In *Programs as Data Objects*.

Nielson, F. 1989. Two-level semantics and abstract interpretation. *Theor. Comp. Sci. 69*, 117–242.

Nielson, F. and Nielson, H. 1992. *Two-Level Functional Languages*. Cambridge Univ. Press.

Pees, S., Hoffmann, A., Zivojnovic, V., and Meyr, H. 1999. LISA machine description language for cycle-accurate models of programmable DSP architectures. In *DAC*.

Pettersson, M. 1992. A term pattern-match compiler inspired by finite automata theory. In *CC*.

Pleban, U. and Lee, P. 1987. High-level semantics. In *Workshop on Mathematical Foundations of Programming Language Semantics*.

PowerPC32. The PowerPC User Instruction Set Architecture. doi.ieeecs.org/10.1109/MM.1994.363069.

Ramalingam, G., Field, J., and Tip, F. 1999. Aggregate structure identification and its application to program analysis. In *POPL*.

Ramsey, N. and Davidson, J. 1999. Specifying instructions' semantics using λ-RTL. Unpublished manuscript.

Ramsey, N. and Ferandez, M. F. 1994. New Jersey machine-code toolkit arch. spec. technical report. Tech. Rep. TR-470-94.

Regehr, J. and Reid, A. 2004. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Architectural Support for Prog. Lang. and Op. Syst.*

Reps, T., Balakrishnan, G., and Lim, J. 2006. Intermediate-representation recovery from low-level code. In *Part. Eval. and Semantics-Based Prog. Manip.*

Reps, T., Lim, J., Thakur, A., Balakrishnan, G., and Lal, A. 2010. There's plenty of room at the bottom: Analyzing and verifying machine code. In *Computer Aided Verif.*

Reps, T., Sagiv, M., and Loginov, A. 2010. Finite differencing of logical formulas for static analysis. *Trans. on Prog. Lang. and Syst. 6,* 32.

Reps, T., Sagiv, M., and Yorsh, G. 2004. Symbolic implementation of the best transformer. In *Verif., Model Checking, and Abs. Interp.* 252–266.

Reps, T., Schwoon, S., Jha, S., and Melski, D. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog. 58,* 1–2 (Oct.), 206–263.

Scherpelz, E., Lerner, S., and Chambers, C. 2007. Automatic inference of optimizer flow functions from semantics meanings. In *PLDI*.

Schmidt, D. 1986. *Denotational Semantics*. Allyn and Bacon, Inc., Boston, MA.

Schmidt, D. 1998. Data-flow analysis is model checking of abstract interpretations. In *Princ. of Prog. Lang.* 38–48.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall.

Siewiorek, D., Bell, G., and Newell, A. 1982. *Computer Structures: Principles and Examples.* Springer-Verlag.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. 2008. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*.

SOOT. SOOT: A Java optimization framework. www.sable.mcgill.ca/soot/.

Steffen, B. 1991. Data flow analysis as model checking. In *Theor. Aspects of Comp. Softw.* Lec. Notes in Comp. Sci., vol. 526. Springer-Verlag, 346–365.

Steffen, B. 1993. Generating data flow analysis algorithms from modal specifications. *Sci. of Comp. Prog. 21,* 2, 115–139.

Thakur, A., Elder, M., and Reps, T. 2012. Bilateral algorithms for symbolic abstraction. In *Static Analysis Symp.*

Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., and Reps, T. 2010. Directed proof generation for machine code. In *Computer Aided Verif.*

Thakur, A. and Reps, T. 2012. A method for symbolic computation of abstract operations. In *Computer Aided Verif.*

Tjiang, S. and Hennessy, J. 1992. Sharlit: A tool for building optimizers. In *Prog. Lang. Design and Impl.*

Venkatesh, G. 1989. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *Prog. Lang. Design and Impl.*

Venkatesh, G. and Fischer, C. 1992. SPARE: A development environment for program analysis algorithms. *Trans. on Softw. Eng. 18,* 4.

Wadler, P. 1987. Efficient compilation of pattern-matching. *The Impl. of Func. Prog. Lang.*.

WALA. WALA. wala.sourceforge.net/wiki/index.php/.

WALi 2007. WALi: The Weighted Automaton Library. www.cs.wisc.edu/wpis/wpds/download.php.

Whaley, J., Avots, D., Carbin, M., and Lam, M. 2005. Using Datalog with Binary Decision Diagrams for program analysis. In *Asian Symp. on Prog. Lang. and Systems*.

Wilhelm, R. 1981. Global flow analysis and optimization in MUG2 the compiler generating system. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall.

Wilson, R., French, R., Wilson, C., Amarasinghe, S., Anderson, J.-A., Tjiang, S., Liao, S.-W., Tseng, C.-W., Hall, M., Lam, M., and Hennessy, J. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices 29,* 12.

WPDS++ 2004. WPDS++: A C++ library for Weighted Pushdown Systems. "http://www.cs.wisc.edu/wpis/wpds++".

Yi, K. and Harrison, III, W. 1993. Automatic generation and management of interprocedural program analyses. In *Princ. of Prog. Lang.*