✓ Artifact evaluated by FSE

Extracting Instruction Semantics via Symbolic Execution of Code Generators[*]

Niranjan Hasabnis[†]
Intel
CA, USA
niranjan.hasabnis@intel.com

R. Sekar
Stony Brook University
NY, USA
sekar@cs.stonybrook.edu

ABSTRACT

Binary analysis and instrumentation form the basis of many tools and frameworks for software debugging, security hardening, and monitoring. Accurate modeling of instruction semantics is paramount in this regard, as errors can lead to program crashes, or worse, bypassing of security checks. Semantic modeling is a daunting task for modern processors such as x86 and ARM that support over a thousand instructions, many of them with complex semantics. This paper describes a new approach to *automate* this semantic modeling task. Our approach leverages instruction semantics knowledge that is already encoded into today's production compilers such as GCC and LLVM. Such an approach can greatly reduce manual effort, and more importantly, avoid errors introduced by manual modeling. Furthermore, it is applicable to any of the numerous architectures already supported by the compiler. In this paper, we develop a new symbolic execution technique to extract instruction semantics from a compiler's source code. Unlike previous applications of symbolic execution that were focused on identifying a single program path that violates a property, our approach addresses the *all paths* problem, extracting the entire input/output behavior of the code generator. We have applied it successfully to the 120K lines of C-code used in GCC's code generator to extract x86 instruction semantics. To demonstrate architecture-neutrality, we have also applied it to AVR, a processor used in the popular Arduino platform.

CCS Concepts

•**Theory of computation → Program analysis;** *Pre- and post-conditions; Abstraction;* •**Software and its engineering → Source code generation; Automatic programming;** *Retargetable compilers; Software reverse engineering;*

[*]This work was supported in part by grants from NSF (CNS-1319137) and ONR (N00014-15-1-2378).

[†]This work was completed while the first author was a PhD student at Stony Brook University.

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950335

Keywords

Instruction-set semantics extraction; Code generators; Symbolic execution

1. INTRODUCTION

Binary analysis and instrumentation play a central role in software monitoring and debugging, as well as hardening of commercial off-the-shelf (COTS) software. Many popular tools and techniques, including those for software emulation and virtualization (e.g., QEMU [7], Valgrind [38], DynamoRio [8] and Pin [36]), malware analysis [47, 9, 19, 54], exploit defense (e.g., taint-tracking [39, 41, 42], control-flow integrity [2, 57, 55]), and sandboxing [34, 20, 53] rely on these techniques.

One of the major challenges in binary instrumentation is the complexity of modern instruction sets. The Intel manual describing the x86 instruction set runs to over 1500 pages describing over 1100 instructions. Modern ARM instruction sets are even larger. Moreover, this semantic modeling task is not a one-time effort, since there are many processors to be considered, e.g., processors used in embedded systems. Even for mature processors, the instruction sets are frequently expanded. As a result, even mature platforms such as Valgrind support only a small number of processors, and omit subsets of instructions such as AVX, FMA4, and SSE4.1 on x86.

Binary analysis and instrumentation present an unforgiving environment for modeling errors: not only are such errors hard to debug, they also cause the instrumented application to fail; or worse, they allow security checks to be bypassed.

In this paper, we present a new approach, called EISSEC (Extracting Instruction Semantics by Symbolic Execution of Code generators), that overcomes these challenges by *automatically synthesizing instruction semantics* from information already contained in existing compilers. Specifically, modern compilers such as GCC and LLVM incorporate a code generator that translates code from a low-level intermediate representation (e.g., RTL in the case of gcc) to assembly. We develop a novel symbolic-execution-based technique to extract the semantics encoded in the source of such a code generator. The extracted semantics maps assembly instructions into the compiler's architecture-independent internal representation (IR). Not only does this approach avoid manual labor, but it also benefits from years of testing and debugging already performed on the code generator. More importantly, the approach is architecture-neutral, i.e., it can support any of the numerous architectures already supported by a compiler such as GCC.

Below, we summarize the structure of code generators in modern compilers, and provide the rationale for the approach developed in this paper. We then identify the key challenges faced in this approach, and summarize our contributions.

1.1 Approach Motivation

Architecture-independent code generation has long been a focus in compiler research. These code generators translate code from an (architecture-independent) intermediate representation (IR) to assembly. Target independence is achieved by driving a generic code generator with *machine descriptions* (MDs) that model the specifics of the target instruction set. The bulk of the MD consists of rules that specify how to translate a snippet of IR into an assembly instruction. These rules are matched up against the IR produced by the compiler front-end, and the matching snippets are then replaced by the corresponding assembly instruction.

A purely rule-based approach can generate inefficient code since it fails to take into account the context in which translation takes place, e.g., it may generate many redundant loads and stores. Davidson and Fraser [17] showed that these inefficiencies can be mitigated by performing several optimizing transformations on the generated code. Moreover, they showed that the MDs can be used to make these optimizations target independent. This approach thus moves the complexity of code generation to IR optimization passes that are shared across different architectures, while simplifying architecture-specific MDs. Code generators in contemporary compilers such as GCC and LLVM follow this general outline. We based our implementation on GCC's code generator due to its maturity, and support for many instruction sets.

To provide a better understanding of the rules used in MDs, consider the following rule:

```
[(set (match_operand:0 "register_operand" "=a")
   (div (match_operand:1 "register_operand" "0" )
     (match_operand:2 "nonimmediate_operand" "qm")))
  (clobber (reg:  FLAGS_REG))]
⟶ "div %2"
```

The first part of the rule specifies the semantics of the `div` instruction in RTL, the IR used by GCC. It indicates that operand 0 contains the result of dividing operand 1 by operand 2. At the assembly level, the first two operands are implicitly the *A*-register, a fact captured by match constraints "**0**" and "**=a**" in the RTL. The applicability of the pattern is constrained by several other (architecture-specific) constraints, including "***register_operand***" and "**qm**." Often, compilers don't need the exact flag values after every arithmetic or logical instruction. Hence this rule indicates that flags are modified without specifying exactly how[1].

It might seem that we can simply use these rules in the reverse direction to translate assembly to IR. However, several major difficulties arise in doing so. We found that key details, such as the meanings of the constraints and the printing of assembly-level operands (i.e., how `%2` is substituted in the above example) are hard-coded into architecture-specific C-functions. Even the output assembly may not be specified for some rules: instead, the right-hand side of the rule may consist of C-code that, when compiled and executed, will return a string representing the output assembly instruction!

[1]The impact of this abstraction is discussed at the end of Section 4.

One possible avenue is to rely on human experts to specify the behavior of all this C-code in a declarative form that is amenable to automated analysis and use in the reverse direction. This was the approach used by DisIRer [33]. Unfortunately, the amount of code that needs to be manually analyzed is substantial. For instance, the x86 MD in GCC consists of 1500 rules, but the amount of x86-specific C-code is about 17K lines! In addition to the scale of effort involved, manual approaches negate two of the key benefits of a compiler-based approach:

- The manual effort will need to be repeated for each architecture. Thus, the approach is unable to benefit much from a compiler's support for many different architectures.
- Manual efforts will invariably introduce errors, there by necessitating extensive testing, thus negating another major benefit of relying on a well-tested code generator.

The above drawbacks of manually reversing a code generator are avoided by the learning-based approach called LISC [30, 29] we developed earlier. In this approach, a variety of binaries were compiled to produce training data (in the form of $\langle assembly, IR \rangle$ pairs) for an algorithm that learns generalized assembly-to-IR mapping rules. An important drawback of LISC is that its design does not ensure either soundness or completeness. Instead, these properties have to be evaluated experimentally.

We therefore present an alternative approach called EISSEC that is based on symbolic execution of the code generator. Instead of blindly generalizing from observed results, EISSEC extracts *the actual mappings contained in the code generator.* Thus, the IR to assembly mappings extracted are sound by design. Moreover, by developing an all-paths exploration technique, we ensure the extraction of the *complete behavior* of the code generator.

1.2 Key Challenges and Contributions

While we want to translate assembly to IR, code generators perform the inverse task: they map IR to assembly. Hence we need to develop techniques for *inverting the translation function* of a code generator. If the code generator was driven by a declarative specification of the mapping, the inversion problem may not be so hard. Indeed, many modern code generators are driven by so-called machine descriptions (MDs) that aim to specify such mappings. However, in practice, the size of target-architecture-specific C (or C++) code used in these code generators is rather large. For instance, GCC's x86 machine description consists of 1500 pattern-matching based rules, together with 17K lines of C-code[2]. Thus, to realize EISSEC, we must solve the following problems:

- *Extracting functions specified in code.* We develop a novel symbolic-execution based approach for extracting IR-to-assembly mapping from a code generator. Symbolic execution has been used successfully in the context of software testing [23, 12, 11, 24, 22, 35, 46], vulnerability analysis [10], malware analysis [43] and exploit generation [5]. However, these applications are generally concerned with the discovery of one program path, or, equivalently, the problem of identifying one ⟨input, output⟩ pair. SAT and SMT solvers excel in discovering such (counter-)examples. In contrast, ours is an *all-paths* problem, one that requires *all*

[2]This excludes all architecture-independent code, e.g., RTL optimization passes and the generic rule-based translation framework.

possible ⟨input, output⟩ pairs. While code generators are large, they are constrained in some ways — for instance, loops are rare, and pointer uses are relatively simple. By exploiting these features, and by carefully engineering a symbolic execution engine over a constraint-logic programming system[3], we have overcome the challenges of all-paths symbolic execution in this domain.

- *Soundness.* MDs are meant to translate a compiler's IR to assembly, so a natural question is whether they can be used backwards to lift assembly to IR. Section 4 provides justification for this inversion.
- *Completeness.* The mapping derived by our approach may be incomplete for two reasons. First, a code generator may not use all of the instructions in the target architecture. Our evaluation shows that this isn't a significant source of incompleteness in practice. Secondly, the semantics may abstract out some details, e.g., the exact value of CPU flags after an instruction. Our evaluation measures the scope of such incompleteness. We also discuss how binary analysis and instrumentation techniques can generally work with this imprecision. (See Section 5.)
- *Efficient implementation.* Our symbolic execution system is efficient and and is able to achieve all-paths execution for the 120K lines of C-code used by GCC in its x86 code generator. In contrast with previous techniques that often limited their focus to a subset of instructions, we handle the entire x86 instruction set, including extensions such as SSE4.1, AVX, and FMA4.
- *Architecture-neutrality.* It took us just about 8 person-hours to extract instruction semantics for AVR, an embedded systems processor used in the popular Arduino platform, as well as many automotive applications.

Paper organization. In Section 2, we describe EISSEC, our symbolic execution system for extracting IR to assembly mapping. Section 3 and section 4 discuss the inversion of these rules, and the soundness of doing so. Experimental evaluation of the completeness, soundness and performance of our approach is described in Section 5. Related work is discussed in Section 6, followed by concluding remarks in Section 7, and artifact description in Section 8.

2. EXTRACTING IR TO ASSEMBLY MAP

A conceptually simple approach for extracting IR to assembly mapping is to generate every possible IR snippet that can be given as input, and observing the output. However, the number of possible snippets is far too large: even a simple instruction set consisting of one opcode and one 32-bit operand will require considering 4 billion possibilities. Instead of eagerly instantiating all possible IR snippets, we develop a *concolic execution*[4] *technique* that relies on lazy instantiation of IR (specifically, RTL patterns). We outline this technique below, and illustrate it with a small example. More details about the approach can be found in the first author's dissertation [26].

2.1 Approach Overview

Symbolic/concolic execution systems such as KLEE [10], DART [23], EXE [12], and CUTE [46] have proven effective in the context of automatic input generation for software testing and bug-finding. The nature of these applications, however, is qualitatively different from that targeted by EISSEC. In particular, although these systems try to exercise many paths in the code, they don't ensure the exploration of all paths in nontrivial programs. More importantly, they don't attempt to generate every possible input that can drive a program down a specific path. For this reason, they are underpinned by SAT/SMT solvers that are optimized to find a single solution to a formula. In contrast, EISSEC needs to extract the *complete input/output mapping,* i.e., it needs to consider every possible input, and compute the corresponding output. Thus, EISSEC needs to:

- traverse *all possible paths* in the code generator, and
- compute *all solutions* to formulas, i.e., solve the all-SAT problem.

For these reasons, we have developed a concolic execution system that is targeted at the function extraction problem.

Extracting the complete input/output behavior of arbitrary programs is infeasible in general. However, code generators are constrained in many ways. For instance, loops are relatively rare, and even when they occur, they can be easily unrolled. Moreover, pointer uses are relatively simple. By exploiting these features, we have developed an effective all-paths/all-solutions concolic execution system that scales to relatively large programs, such as the 120KLoC x86 code generator used by GCC. For simplicity, we describe our approach using a hypothetical code generator for the following (toy) instruction set.

$$\begin{array}{rcl} \mathit{Instruction} & ::= & \texttt{mov}\ S, D \mid \texttt{add}\ S, D \\ S & ::= & \texttt{r}N \mid (\mathit{Int}) \mid \texttt{\$}\mathit{Int} \\ D & ::= & \texttt{r}N \mid (\mathit{Int}) \\ N & ::= & 1..10 \end{array}$$

Assume that the processor imposes the following restrictions on the operands of these instructions:

1. stores (moves to memory) must be from registers
2. loads can only target registers 1 through 3.
3. add instruction accepts only registers arguments; moreover, the source register number should be the square root of the destination register. This odd restriction is introduced as an example of a constraint that won't be supported by a constraint solver. It will need to be handled by enumerating possible values of a symbolic variable.

Figure 1 shows a code generator for this instruction set, implemented by a function `recog`. It takes a single argument `I` which is an RTL snippet, represented using a datatype `rtx`. It prints an assembly instruction and returns 0 when there is a valid translation for `I`, and returns −1 otherwise. It is written in C, except that we have (a) used indentation rather than braces to capture block nesting, and (b) omitted the local variable declarations. Each of the above three cases is handled by the blocks of code beginning respectively on lines 4, 11 and 20 in this figure.

[3]Logic programming systems are generally designed to produce most-general solutions to constraints, and to generate all solutions when needed — the reasons we have relied on a CLP(fd) constraint solver (rather than a more conventional SAT/SMT solver) in our implementation.

[4]Concolic execution stands for mixed concrete and symbolic evaluation, where some of the variables assume symbolic values, while the remaining ones assume concrete values.

```
int recog(rtx i)
  if(GETCODE(i)==SET)
  s=XEXP(i, 2); d=XEXP(i, 1);
    if(GETCODE(s)==REG &&
      (GETCODE(d)==MEM||GETCODE(d)==REG))
        print("mov ");
        printop(s);
        print(", ");
        printop(d);
        return 0;
    else if(GETCODE(d)==REG)
      n=GETREG(d);
      if(GETCODE(s)==REG||GETCODE(s)==IMM)
        || (GETCODE(s)==MEM && n<4))
          print("mov ");
          printop(s);
          print(", ");
          printop(d);
          return 0;
      else if(GETCODE(s)==ADD)
        r1=XEXP(s, 1);
        r2=XEXP(s, 2);
        if(GETCODE(r1)==GETCODE(r2)==REG)
          n1=GETREG(r1); n2=GETREG(r2);
          if(n2==n && n1==sqrt(n2))
            print("add ");
            printop(r1);
            print(" ");
            printop(d);
            return 0;
  return -1;
```

```
int recog_tr(rtx i, rtx_meta i_meta, rtx_meta ret_meta)
  if try( GETCODE_CONS(i, SET)) // constraint: I = set(X,Y)
  XEXP_CONS(i, 2, s); // effective constraint: S = X
  XEXP_CONS(i, 1, d); // D = Y
  GETCODE_CONS(s, c1); GETCODE_CONS(d, c2);
  if try(c1==REG && (c2==MEM || c2==REG))
    print_tr(o, "mov"); printop_tr(s);
    print_tr(o, ", "); printop_tr(d);
    record_mapping(i, o);
    add_return_cons(ret_meta, 1); backtrack();
  else if try(c2==REG) // D = reg(_)
    GETREG_CONS(d, n); // D = reg(N)
    if try(c1==REG||c1==IMM||(c1==MEM && n<4))
      print_tr(o, "mov ");
      printop_tr(s);
      print_tr(o, ", ");
      printop_tr(d);
      record_mapping(i, o);
      add_return_cons(ret_meta, 1); backtrack();
    else if try(c1==ADD) // S = add(_, _)
      XEXP_CONS(s, 1, r1); XEXP_CONS(s, 2, r2);//S = add(R1, R2)
      GETCODE_CONS(r1, c5); GETCODE_CONS(r2, c6);
      if try(c5==REG && c6==REG) //R1 = reg(_), R2 = reg(_)
        GETREG_CONS(r1, n1); GETREG_CONS(r2, n2);
        if try(n2==n && n1==sqrt(n2)) // N2 = N
          while (getNext(n2, &n3))
            n1=sqrt(n3);
            print_tr(o, "add ");printop_tr(r1);print_tr(o, ", ");
            printop_tr(d); record_mapping(i, o);
            add_return_cons(ret_meta, 1); backtrack();
  add_return_cons(ret_meta, -1), backtrack();
```

Figure 1: A code generator for the toy instruction set (left), and its tranformation for concolic execution (right).

2.1.1 *Code Transformation for Concolic Execution*

Our concolic execution engine is implemented as a source-to-source transformation on C-code, implemented using CIL-1.4.0 [37]. GCC's code generator[5] also contains the `recog` function shown in the toy example. Concolic execution will begin with a call to `recog` with a symbolic argument. Other than this argument, all remaining objects in memory will be concrete at this point. As `recog` executes, it will begin to assign symbolic values to more variables. We use a simple static analysis to identify functions that could be called by `recog` with symbolic arguments, and the transformation is recursively applied to those functions.

For performance and scalability, the number of symbolic variables should be kept as small as possible. To accomplish this, the transformed code incorporates runtime checks on whether variables flagged by our static analysis to be symbolic are indeed symbolic at runtime, and if not, switch to using concrete computations on them.

To ensure consistency between symbolic and concrete state, we limit pointers from being flagged as symbolic, except in cases where the pointer can be treated as a opaque handle to an abstract data type. In addition, global variables are treated as concrete unless otherwise indicated using annotation. In our case, we had to annotate 3 of GCC's global data structures that store `rtx` pointers. Array elements and structure fields accessed using constant offsets are permitted to hold symbolic values.

After marking variables as symbolic or concrete, code transformation, as illustrated in Figure 1, is carried out. First, we model `rtx` objects as first-order terms, and transform calls to its accessor functions into term constraints. For the toy example, the structure of this term is given by:

$$rtx = set(X, Y)|add(X, Y)|reg(n)|mem(n)|imm(n)|int(n)$$

Our constraint solver is a logic programming system, so we use the Prolog convention that variables start with upper-case, while constants begin with lower-case letters. The code generator, written in C, uses the opposite convention: variables are lower-case, while constants are upper-case.

Calls to accessor functions such as `GETCODE` get replaced by a call to a symbolic version `GETCODE_CONS`. These symbolic versions are developed manually, but their number is small, and moreover, they are very simple and architecture-independent.

If the value returned by an accessor function is used in a comparison, for instance, `GETCODE(I)==SET`, the transformed version generates a corresponding constraint, namely $I = set(X, Y)$. (Here X and Y are new symbolic variables.) This constraint $\mathcal{C}$ is handed to a runtime function **try** that forwards it to the constraint solver. If the solver indicates that $\mathcal{C}$ can either be true or false, then **try** causes the symbolic execution engine to fork[6], and explore both branches. This is done in concert with the constraint solver's backtracking mechanism. However, if the solver indicates the only of the branches is feasible, **try** won't fork.

In the case of assignments, we ensure that the variable being assigned is a fresh variable, so there is no possibility of the constraint failing. So, the constraint is simply added on the current path, without forking. Thus, the statement `s=XEXP(i,1)`, which is transformed to `XEXP_CONS(i,1,s)`, ends up generating the constraint $I = set(S, \_)$, where we use the Prolog convention of "_" to denote a "dont-care" variable.

[5]We used GCC-4.6.4, which was the most recent version of GCC at the time we began our project.

[6]We use a light-weight version of fork, as described later.

Arithmetic, logic, and comparison operations on symbolic variables will result in the generation of a constraint that is handed over to a constraint solver. In addition, output generation is also handled using constraints. The output is treated as a term — specifically, a list whose elements get defined as the print operations are executed, but the tail remains a symbolic variable until `recog` returns. At this point, if the return code is negative, we generate the `false` constraint that causes the currently explored path to fail. Otherwise, we terminate the output list, and record the mapping between input and output. The constraint solver is queried to extract the contents of input and output at this point, based on all the constraints processed by it so far.

2.1.2 Constraint Solver

Even though SMT solvers [21, 18] are commonly used by symbolic execution systems [10, 46], they are typically engineered to produce single solutions very quickly, but not so much for the all-solutions problem [32, 51]. Logic programming systems, on the other hand, have been optimized for systematic and efficient enumeration of all solutions. Specifically, we use a logic-programming system with a finite-domain constraint solver (CLP(fd)) in EISSEC.

Unlike term/structure constraints supported in plain logic programming, CLP(fd) systems support a wide range of constraints. Unfortunately, as a result, it is not possible to keep the constraints in the most simplified form, i.e., the equivalent of most general unifiers either don't exist, or cannot be efficiently computed incrementally, as constraints are added. Nevertheless, these systems incorporate a primitive to systematically instantiate variables over a finite-domain, and to return those values that are consistent with the current set of constraints. We rely on this primitive, especially to handle complex constraints such as the square-root constraint in the toy instruction set.

2.1.3 Mapping Extracted for Figure 1

The first case in the code generator (lines 2 to 10) corresponds to two execution paths, one where the destination operand of memory, and the other with a destination register. These two paths yield the following rules:

$$\begin{array}{rcl} set(mem(X), reg(Y)) & \longrightarrow & mov\ \ rY, (X) \\ set(reg(X), reg(Y)) & \longrightarrow & mov\ \ rY, rX \end{array}$$

The next block (lines 11 to 19) contains three disjuncts, thus yielding the following three rules.

$$\begin{array}{rcll} set(reg(X), reg(Y)) & \longrightarrow & mov\ \ rY, rX & \\ set(reg(X), mem(Y)) & \longrightarrow & mov\ \ (Y), rX, & X < 4 \\ set(reg(X), imm(Y)) & \longrightarrow & mov\ \ \$Y, rX & \end{array}$$

Finally, the last block (lines 20 to 30) contains no disjunctions, but the constraint solver needs to enumerate the possible register numbers to generate suitable values. Since there are 10 possible register values, three combinations are possible, thus yielding 3 rules.

$$\begin{array}{rcl} set(reg(1), add(reg(1), reg(1))) & \longrightarrow & add\ \ r1, r1 \\ set(reg(4), add(reg(2), reg(4))) & \longrightarrow & add\ \ r2, r4 \\ set(reg(9), add(reg(3), reg(9))) & \longrightarrow & add\ \ r3, r9 \end{array}$$

It is important to note that our approach generates parameterized rules. This is achieved by permitting the output to be a list, where some elements could be variables. By associating these variables names between the input and output, we are generating parameterized rules.

2.2 Implementation

2.2.1 Source-to-source Transformation

Source-to-source transformation is implemented as a plug-in to CIL [37], a popular open-source transformation system. The implementation of the plug-in consists of around 2600 lines of OCaml code. The plugin utilizes some of the CIL features such as a simplification pass to generate 3-address code from C code to simplify the transformation. It also handles other challenges such as avoiding the transformations of system code included from header files. This is done by getting a list of directories which are included in the compiler's include search path. We treat files included from any of those directories as a header files and avoid their transformation.

2.2.2 Concolic Execution Engine

The transformer needs support code (written in C) to implement various functions such as `addOpCons` etc. This support code is approximately 7200 lines. It also includes approximately 3000 lines of C code for RTL accessor functions. Note that this code does not have architecture dependencies.

Although we limit the propagation of symbolic values for various pragmatic reasons, these limitations do not pose hard constraints in terms of code constructs that can be handled. For instance, consider the case of a symbolic value X passed to an external function that hasn't been transformed (and hence can accept only concrete input values). At this point, our concolic execution engine invokes the constraint solver to enumerate all possible (concrete) values of X that are consistent with the current set of constraints accumulated on X. Then it forks itself once for each such value, and then calls the external function with that value.

Our undo record approach is implemented in around 500 lines of C++ code. The space complexity of our approach is proportional to the length of the program path (in terms of branch points). Thus, EISSEC can scale to millions of paths.

2.2.3 Constraint Solver

The implementation of our constraint solver uses SWI-Prolog [52], a popular Prolog engine, with its support for CLPFD [48]. GCC's code generator only generates finite-domain constraints, so CLPFD is enough for our purpose. The solver is implemented in around 700 lines of Prolog and is supported by 1000 lines of C code. It uses the well-known Prolog concept of backtracking (using `fail.`) and other supported features of SWI-Prolog such as association lists, enumerating all solutions to a query using `labeling`.

We execute the constraint solver as a stand-alone process separate from the process executing the transformed C code. The support library provides functions used by the transformed code to interact with the solver.

Although, we could map most of our requirements from the constraint solver into the predicates of CLPFD or SWI-Prolog, one problem demanded special treatment. Specifically, we found that neither CLPFD nor SWI-Prolog provides predicate(s) to access the set of constraints imposed on the variables. To solve this problem, we access the constraints using `clpfd_attr` and access its propagators via `fd_props`[7]

[7]These are predicates internal to SWI-Prolog.

as SWI-Prolog stores the constraints using attributed variables [49]. In order to capture a complete set of constraints on all the input and output variables of a mapping rule, we traverse the dependence graph of the variables starting from the output variables and reaching all the input variables. The dependence graph is a graph where the nodes are variables and the edges are constraints between the variables. The goal of the traversal is to print constraints appearing on all the paths between the output and the input variables.

2.2.4 *Optimizations*

Symbolic to concrete conversion ("enumeration" as we described earlier) leads to a large number of program paths, so we have implemented several optimizations to limit it:

- *Using range and set constraints.* This optimization relies on set and range constraints supported by SWI-Prolog engine. Instead of generating a concrete process for every concrete enumerated value, it uses set or range constraint to generate only one concrete process. In symbolic execution context, this optimization achieves the same effect as that of merging paths. We implement this optimization in array accessor functions, and in case of GCC, some of the arrays contain hundreds of elements. By using range constraints, such elements could be represented using a few ranges.
- *Exploiting hardware-level parallelism.* As mentioned earlier, parallel exploration of paths could be pursued with significant benefits at the higher levels of the search tree, while at the lower levels, benefits are less significant. We have implemented a simple parallelization strategy that reverts to the use of efficient (but serial) undo mechanism at lower levels, while relying on forks at the higher levels.

3. LIFTING BINARIES TO IR

As described earlier, our concolic execution generates parameterized rules of the form $IR \longrightarrow Asm$. The example rules for the toy instruction set was shown in Section 2.1.3. The key idea is to use these rules in reverse in order to lift binaries to IR. While conceptually simple, several additional problems need to be addressed in order to apply this approach to whole binaries:

- *Disassembly:* For stripped binaries, disassembly can be a nontrivial task. However, recent works (e.g., [57, 56, 55]) have developed solutions that are robust enough to scale to large binaries, and we simply build over these solutions.
- *Efficient lookup:* For large instruction sets, the technique described in the last section can generate millions of rules. To use this rule set efficiently, we construct a matching automaton from these rules. Specifically, we parse assembly instructions to construct ASTs, and the matching automaton operates on this AST. Efficient tree automata construction techniques have been well-studied [44, 45], and their use in assembly-to-IR lifting is also known [30].
- *Handling one-to-many mapping:* Multiple IRs may be translated into the same assembly instruction. As a result, when the mapping is inverted, a single assembly instruction may map to multiple IRs. As we discuss later, these mappings are all sound, but some may be more precise than others, e.g., contain fewer "clobber" declarations. EISSEC chooses more precise translations when available.
- *Handling many-to-one mapping:* There are instances when the code generator maps an IR snippet into a sequence of assembly instructions. As a result, a simple approach that lifts a single assembly instruction at a time won't always work. A naive approach for handling such many-to-one translations can lead to exponential complexity. In previous work [30], we developed a linear-time dynamic programming technique to handle many-to-one mappings.
- *Soundness of reversing IR to assembly mapping.* This is perhaps the most important consideration, and is the topic of the next section.

4. SOUNDNESS

As described earlier, MDs are used to translate RTL to assembly, so a natural question is whether it is sound to use them backwards. There is an important practical reason to believe this: from a developer's perspective, an MD is developed by enumerating instructions in the target architecture, and specifying the equivalent RTL.

The second, and more important reason for soundness stems from how these rules are used in code generators. As discussed before, these MD pattern-matching rules are applied against IR generated from the compiler front-end. Any time there is a match, a rule can be used. This essentially means that in every system state, the behavior of IR and assembly code components of any rule must have close correspondence. Otherwise the assembly instruction will either fail to achieve a required effect (e.g., fail to set a result register to a specified value), or have spurious additional effects (e.g., modify a register that was indicated as being preserved in the IR). Intuitively, this means that the semantics of IR and assembly should be in close correspondence. We begin with a definition of this correspondence, taken from [27].

The state $\mathbf{S}_{asm}$ at the assembly level is given by an assignment of values to a set of variables V_{asm} representing the processor's internal registers, memory, etc.

The state $\mathbf{S}_{RTL}$ at the RTL-level is given by an assignment of values to a set of variables V_{RTL} representing the processor's state as viewed by the code generator. We assume that $V_{RTL} \subseteq V_{asm}$. The set of permitted values for a variable v at the RTL level will include all of the values permissible at the assembly level, plus a special value called $\top$ that captures the idea that the value is unknown or undefined.

$\mathbf{S}_{RTL}$ is said to be ***valid*** for an RTL snippet R if for every variable v read by R (excluding those variables that are first updated and then read by R), $\mathbf{S}_{RTL}(v) \neq \top$.

DEFINITION 1 (PROCESSOR STATE CORRESPONDENCE [27]). *States $\mathbf{S}_{asm}$ and $\mathbf{S}_{RTL}$ are said to correspond if:*

$$\forall v \in V_{asm}\ (\mathbf{S}_{RTL}(v) = \mathbf{S}_{asm}(v)) \vee (\mathbf{S}_{RTL}(v) = \top)$$

This definition says that $\mathbf{S}_{RTL}$ *is a conservative approximation of* $\mathbf{S}_{asm}$: either they agree on the value of a state variable, or $\mathbf{S}_{RTL}$ leaves it unspecified. The latter choice is made when a developer specifies that an instruction "clobbers" something, without specifying an exact value. This is mostly done for CPU flags.

While the state correspondence notion seems very reasonable, note that it does imply that IR state should be mapped to concrete processor state. For instance, virtual registers need to be mapped to concrete registers. This holds in the case of GCC, the compiler used in our implementation.

Based on the correspondence between the *states* at the IR and assembly levels, we can define a notion of correspondence between *instruction behaviors* at the assembly and IR levels. We use the notation $R : S \longrightarrow S'$ to denote that the execution of R in state S leads to a new state S'.

Definition 2 (*recog* soundness). *Let R be such that recog(R) yields the assembly instruction A. Let $\mathbf{S}_{RTL}$ be any state that is valid for R, and $\mathbf{S}_{asm}$ be any corresponding state. Let $R : \mathbf{S}_{RTL} \longrightarrow \mathbf{S}'_{RTL}$ and $A : \mathbf{S}_{asm} \longrightarrow \mathbf{S}'_{asm}$. The rule $R \longrightarrow A$ produced by recog is sound if $\mathbf{S}'_{RTL}$ and $\mathbf{S}'_{asm}$ correspond to each other.*

Note that this definition captures the intuition that an assembly instruction A must do every thing that is done by the corresponding RTL instruction R, but it may also do "more." In particular, from the definition of state correspondence, $\mathbf{S}'_{asm}$ may differ from $\mathbf{S}'_{RTL}$ in terms of (a) state variables that are not captured at the RTL-level, and (b) state variables that have been assigned $\top$ in $\mathbf{S}'_{RTL}$.

Theorem 3 (Code generator soundness). *Any violation of recog soundness will cause the code generator to produce incorrect code.*

Proof: If *recog* soundness is violated, then there must exist an RTL R and corresponding states $\mathbf{S}_{RTL}$ and $\mathbf{S}_{asm}$ such that $recog(R) = A$, $R : \mathbf{S}_{RTL} \longrightarrow \mathbf{S}'_{RTL}$ and $A : \mathbf{S}_{asm} \longrightarrow \mathbf{S}'_{asm}$, but $\mathbf{S}'_{RTL}$ and $\mathbf{S}'_{asm}$ don't correspond. From the definition of state correspondence, this means that there is a state variable $v \in V_{RTL}$ such that $\mathbf{S}'_{RTL}[v] \neq \mathbf{S}'_{asm}[v]$, where $\mathbf{S}'_{RTL}[v] \neq \top$. Now, consider an RTL snippet R' that depends on v. Let $recog(R') = A'$. Thus, the code generator will generate $A; A'$ from RTL snippet $R; R'$. Note that the behavior of $R; R'$ and $A; A'$ cannot be identical because R' starts with a different value of the variable v as compared to A'. ■

Note that there are two implicit assumptions in the proof: that an R' that depends on v can be found, and that there will be some source program whose translation can result in the sequence $R; R'$. It is conceivable that the code generator is based on some deep knowledge that such a combination is impossible, but it does not seem likely; moreover, building in such unspecified assumptions is not a reasonable practice.

Theorem 4 (Soundness of Asm-to-RTL Mapping). *Let R be an RTL snippet such that recog(R) = A. If recog is sound, then R is a sound (i.e., conservative) approximation of the behavior of A. Formally, let $A : \mathbf{S}_{asm} \longrightarrow \mathbf{S}'_{asm}$, and $\mathbf{S}_{RTL}$ be obtained by restricting $\mathbf{S}_{asm}$ to V_{RTL}, and $R : \mathbf{S}_{RTL} \longrightarrow \mathbf{S}'_{RTL}$. Then $\mathbf{S}'_{RTL}$ is a conservative approximation of $\mathbf{S}'_{asm}$.*

Proof: Follows directly from the definition of *recog* soundness. Note that $\mathbf{S}_{asm}, \mathbf{S}'_{asm}, \mathbf{S}_{RTL}$ and $\mathbf{S}'_{RTL}$ satisfy Definition 2 and hence states $\mathbf{S}'_{asm}$ and $\mathbf{S}'_{RTL}$ are in correspondence. Thus $\mathbf{S}'_{RTL}$ is a conservative approximation of $\mathbf{S}'_{asm}$. ■

If multiple RTL snippets $R_1, ..., R_n$ are mapped to the same instruction A, then this theorem implies that each R_i must be a conservative approximation of the semantics of A.

Discussion. Note that a sound translation can be imprecise. In particular, there may be cases where some details of instruction semantics are not captured in the RTL translation. In practice, this mainly occurs in the case of CPU flags. Many arithmetic instructions modify these flags, but the RTL specification of these instructions may only carry the information that flags are clobbered by these instructions. This does not normally pose a problem since compiler-generated code uses CPU flags only after certain specific instructions, such as comparison. Flags resulting from arithmetic operations are not used. As long as the binary code being translated satisfies this condition, use of approximate semantics will pose no problem. Even otherwise, static analysis techniques are generally based on lossy approximations, so should be able to cope with this kind of precision loss. In an instrumentation context, there can be a problem since the lifted RTL may include snippets that access $\top$-valued variables. This problem can also be handled: we keep track of the assembly instructions A that lose precision during the lift-up to their IR R, and moreover, are followed by other instructions that depend on variables V_u left unspecified by R. When the binary is regenerated after instrumentation, we ensure that R is replaced by A, thus ensuring that the values of variables in V_u will remain the same as in the original program.

Thus, any precision loss introduced by EISSEC does not pose a problem for binary instrumentation, as well as most binary analysis techniques. But for other applications, e.g., binary retargeting, this precision loss may not be acceptable. To support such applications, the semantics derived by EISSEC needs to be manually augmented. Even so, EISSEC can greatly reduce manual effort, as it leaves only a small fraction of state variable values unspecified.

5. EVALUATION

In this section, we evaluate EISSEC in terms of its performance on the x86 instruction set (Section 5.1), the completeness of this model (Section 5.2), and EISSEC's ability to support other architectures (Section 5.3). Where appropriate, we compare the performance of EISSEC with the learning-based LISC approach [30]. All evaluations were performed on a 1.90GHz Intel Core i7-3517U processor with 4GB of RAM running Linux 3.13.0-53.

5.1 Performance on x86 Code Generator

When GCC is compiled for an architecture, it uses the machine description for that architecture, together with architecture-specific C-code, to generate a *decision tree.* The decision tree is the analog of the *recog* function in Fig. 1, i.e., it translates RTL snippets to assembly. Our implementation is based on GCC-4.6.4, the same version used in LISC [30].

In the case of x86, the machine description contained 2244 templates (rules), each for translating a snippet of RTL to assembly. These templates cover all of the advanced x86 extensions such as SSE and AVX. The templates themselves may contain C-code. In addition, there is about 17KLoC of architecture-specific C-code in the x86-code generator. From all this code, GCC generated a decision tree consisting of approximately 120K lines of C-code. EISSEC transforms and performs concolic execution on this 120KLoC.

Unoptimized performance. Fig. 2 shows the progress of concolic execution over time. The first column shows the time, while the second and third columns refer to the number of successful and failure paths explored by the concolic execution engine. The fourth column shows the coverage obtained, expressed in terms of the fraction of the leaves in the decision tree that have been visited. The fifth column shows the virtual memory use.

Total Time (Days)	Success Paths (M)	Failure Paths (M)	Coverage (%)	Virtual Memory Use (MB)
0.00	0	0	0	17
0.51	4.5	18.1	13	20
0.89	5.8	28.4	15	20
1.27	7.1	38.6	17	20
1.65	7.3	49.9	17	21
2.04	7.6	61.2	19	20
3.17	8.5	94.9	21	18
4.88	9.0	127.6	37	19
6.09	11.2	156.3	49	19
7.31	17.4	223.3	66	18
8.53	17.6	248.0	69	19
9.75	18.1	268.5	70	19
10.42	19.2	287.5	79	19
11.55	23.5	316.9	87	18
12.22	24.3	336.0	91	18
13.11	25.6	360.2	100	17

Figure 2: Unoptimized performance on x86.

Although 13 days may seem like a significant length of time, what is more remarkable is that, ultimately, 100% coverage is obtained. In particular:

- all possible paths in the 120KLoC of decision tree code have been traversed, *and,*

- all possible ⟨RTL, assembly⟩ pairs corresponding to each of these paths have been generated.

Note that each positive path represents a successful translation of some RTL snippet to assembly. Although the machine description itself contains only thousands of translation templates, the number of success paths traversed by the concolic execution engine numbers in tens of millions. This is because the engine needs to consider all possible input/output mappings that might result from each such template. At one extreme, a naive enumeration of all possible operand values can easily lead to the generation of 10^{10} to 10^{30} input/output combinations for a single template, depending on the number of operands involved. Clearly, our constraint solver is performing much better, which indicates that it is able to avoid enumeration in most cases. Some case where it can't avoid enumeration are as follows. An instruction template may work with many different operand combinations, but in the assembly representation, these combinations differ significantly in their syntax, causing distinct mappings to be computed. Secondly, registers are identified using numbers in RTL, while they have names such as `eax`. Moreover, these names differ, depending on the width of data involved. Each of these factors has a multiplicative effect, thus contributing very quickly to thousands of mappings for each template.

Note that the number of failure paths is far larger than success paths. This is because each successful path performs a series of checks on RTL. Under normal operation, almost all these checks would succeed, but the concolic execution engine does not know this, and hence needs to generate inputs that fail each of those checks. Thus, the number of failure paths would be of the order of $\sum |P_i|$ where $|P_i|$ denotes the length of the ith successful path.

Note that memory use is small, which means that the exploration can be parallelized easily.

Performance with optimizations. Early on in the project, a lot of effort went into building a highly efficient concolic execution engine. This included the development of very light-weight engine-level fork operations that avoided an OS-level fork, but instead relied on application-level maintenance of undo-records. Before these efforts, the engine could handle instruction sets that were orders of magnitude smaller than x86. Only a limited effort has been put into other aspects of optimization, such as range constraint optimization and parallelization discussed in Section 2.2.4.

	Base	+ Range constraints	+ Naive parallelization
CPU time (days)	13.11	9.20	6.64
Success Paths (M)	25.60	10.90	6.32
Failure Paths (M)	360.20	259.10	257.56
Total Paths (M)	386.00	270.00	263.88

Figure 3: Effect of optimizations for x86.

Figure 3 shows performance improvements obtained using the optimizations from Section 2.2.4. Range constraint optimization is quite effective, yielding over 40% performance improvement.

The nature of concolic execution provides almost unbounded opportunities for parallelization. However, to fully exploit it, significant engineering effort needs to be expended on the concolic execution engine. In particular, we need a seamless way to switch between our application-level forks and OS-level forks. Since we have not spent this effort, we opted for a simpler approach that simply divides the decision tree at the top level into several pieces, with each subtree explored serially. Although there were 6 subtrees at the top level, they were not balanced in size, and hence we were able to gain only a 36% improvement. With additional effort, it should be possible to reduce the execution time linearly with the number of available processors.

Performance comparison with LISC. LISC [30] was trained with just over 3M ⟨RTL, assembly⟩ pairs from about a dozen carefully chosen software packages. Since the number of pairs considered is about two orders of magnitude smaller than the total paths considered by EISSEC, it is no surprise that LISC is much faster than EISSEC. However, note that semantics extraction is a one-time effort, so performance is not critical. Factors such as soundness and completeness are far more important, and we discuss these in the next section.

5.2 Code Generator Completeness

Symbolic execution, by design, should extract the complete behavior of the code generator, as long as the coverage achieved is complete — specifically, if we ensure that all possible inputs have been considered. As indicated by our experiments, EISSEC does achieve complete coverage for the x86 code generator. Thus, what is left to evaluate is the *completeness of the code generator*. No additional incompleteness is introduced by our approach.

To evaluate the completeness of GCC's code generator, we used the semantic model extracted by EISSEC to lift the instructions in all of the binaries that ship by default with Ubuntu 14.04 (desktop version). We were able to translate 99.66% of the assembly instructions without any manual effort. The remaining 0.34% corresponded to 47 of the 1187 instructions supported by x86[8]. Many of these missing instructions are either inserted by the assembler, e.g., `enter` and various versions of `nop`, or are low-level instructions (e.g., `cpuid`, `invpcid` and `rdtsc`) that may be found in hand-coded assembly. Also missing are some rarely used arithmetic instructions such as those operating on binary-coded decimals, and those for directly setting/clearing CPU flags.

We did not undertake a formal comparison with Valgrind [38], but do point out that EISSEC supports all of the advanced x86 instructions such as FMA4 that are missing in Valgrind and similar systems.

In comparison with EISSEC, LISC [30] achieved 99.49% coverage on the Ubuntu test. It should be noted that LISC achieved this coverage using a carefully selected training set that was optimized for this specific dataset. Despite this, the number of instructions LISC can't translate is 50% higher than EISSEC. These misses covered two instructions `rcl` and `rcr`, as well as operand modes and combinations for other instructions that weren't present in the training set.

More importantly, soundness of the extracted semantics has been established for EISSEC, but in the case of LISC, soundness is not ensured. While testing techniques have been developed to compensate for this shortcoming, only a small subset of x86 instructions have been tested in [30].

5.3 Architecture Independence

We then followed the same evaluation steps to extract instruction semantics for the AVR processor used in the popular Arduino embedded system platform. AVR is also widely used in automotive applications. We used `avr-gcc-v4.8.2` code generator in our evaluation.

The AVR architecture has 76 mnemonics, and the code generator used for the symbolic execution contained 98 RTL-to-assembly mapping pairs. The decision tree used by the code generator takes approximately 12K lines of C code.

Our source-to-source transformer places some restrictions on the code. Some manual effort is required to modify the code generator to ensure conformance with these restrictions. This manual effort required approximately 7 hours. In comparison, LISC took roughly half this time (3.5 hours). Being a black-box approach, LISC requires less manual effort than EISSEC.

Runtime performance of EISSEC on the AVR decision tree is shown in Fig. 4. All of the optimizations discussed in Section 2.2.4 were used, including 2-way parallelization at the top-level of the decision tree. Note that the number of paths explored is about three orders of magnitude smaller than that of x86. Consequently, model extraction is much faster as well. Four of the 76 AVR instructions (`break`, `nop`, `wdr`, and `sleep`) were not supported by GCC. EISSEC was able to extract the semantics for the remaining 72 instructions.

Total Time (Mins)	Success Paths (K)	Failure Paths (K)	Coverage (%)	Virtual memory (MB)
0	0	0	0	3.2
20	2.1	9.1	13	6.0
40	7.0	29.2	49	8.3
60	7.8	48.5	71	7.3
87	9.7	63.4	100	5.8

Figure 4: Performance on AVR.

6. RELATED WORK

6.1 Binary Analysis and Instrumentation

Most previous binary analysis/instrumentation systems, including DynamoRio [8], Pin [36], QEMU [7], Valgrind [38], SecondWrite [4], CodeSurfer [6], UQBT [16] and many other systems [15] require a hand-written target instruction specification to drive the translator.

Approaches have been developed [13] for assembly-to-IR translation by relying on QEMU's support for multiple architectures. Specifically, they have written a backend for QEMU to translate QEMU's IR to LLVM's IR. BAP [9], on the other hand, directly uses Valgrind's assembly to IR translator. These methods thus inherit any completeness issues from QEMU and Valgrind, which manifest as (a) support for only the most commonly used platforms, and (b) missing support for new and advanced instruction sets.

DisIRer [33] and Dagger [1] are two efforts that use compiler infrastructures to lift binaries to an IR. Dagger relies on the LLVM infrastructure, but their approach is a manual one: it requires a good understanding of LLVM internals and considerable amount of additional code development. DisIRer's goals are similar to ours: using MDs in reverse to lift binaries. However, as discussed earlier, there are many parts of MDs that are not specifications, and the only way to invert them is if we understand the C-code involved, and manually write functions to invert them.

LISC [30, 29] relies on generating abstract mapping rules by learning the assembly-to-IR translations from concrete mapping rules obtained from code generator logs. To generate the logs, we compile a number of source code packages, and record the ⟨IR, assembly⟩ pairs produced. Being a black-box approach, it requires less effort than EISSEC to support each addition architecture. LISC is also faster. However, LISC does not guarantee the completeness of the extracted map. EISSEC, on the other hand, explores all of the code generator paths and guarantees the completeness of the extracted map. More importantly, we prove the soundness of EISSEC, but the soundness of LISC mappings need to be evaluated experimentally (specifically, using test cases). Both LISC and EISSEC were the results of the first author's PhD research, and a fuller discussion and comparison of the two approaches can be found in his dissertation [26].

6.2 Constraint Solvers

SMT solvers, such as STP [21] and Z3 [18], are used in many popular symbolic execution systems [46, 23, 10, 12, 11, 24, 40, 14]. SMT solvers, with support for various theories such as arrays, match closely with the semantics of C language. So, it is no surprise that they form the decision procedure backend for symbolic execution systems.

SAT/SMT solvers are engineered to efficiently find one solution to a given formula. Fortunately, in the context of bug finding or software testing, the domains where these solvers are mostly used, producing one solution is typically enough. In other words, existing solvers are not targeting

[8]This count excludes privileged instructions that cannot be used at the user-level. More details about this dataset, and how we counted the instructions, can be found in [30].

all-SAT problem, i.e., finding *all* solutions to a SAT problem. EISSEC, however, requires the extractions of the complete input/output behavior of a code generator. This requires the identification of every ⟨input, output⟩ pair that can be produced by the code generator. In other words, EISSEC needs to solve an all-SAT problem. Unlike SAT-solvers that are optimized for finding one solution, logic programing systems are designed so that they can produce all solutions if needed. This is why we based EISSEC on an CLP(fd) system.

6.3 Function Extraction

Symbolic execution has been used in the context of program verification. For instance, work presented in [3] uses symbolic execution in order to extract and verify cryptographic protocol models from their C implementations. Although the high-level idea of using symbolic execution to extract a function from C code is similar to EISSEC, there are number of differences. Specifically, the complexity of cryptographic code handled in this work is several orders of magnitude less (100X) than GCC's code generator. Moreover, their system confines symbolic execution to main path in the code and can only handle the protocol implementations with no significant branching. In order to reduce the complexity of symbolic execution, they manually build semantic models for commonly used cryptographic function. EISSEC, on other hand, would use concrete execution in such cases. Approach described in [50] uses model extraction of GUI programs from hand-held devices and compares extracted models with the expected models (obtained from specifications). Their work is mainly concerned with extracting models to capture how system responds to user inputs. Consequently, their model is a state machine which captures system transitions on various event inputs. Definition of model for EISSEC, on the other hand, is simply a mapping between input (RTL) and output (assembly instructions).

Synthesizing functions using input/output samples could be considered as an alternative to function extraction using symbolic execution. Given the complexity of symbolic execution engines, encoding the instruction semantics in symbolic formulas is an interesting challenge. There are existing approaches [25, 31] that use program synthesis or template-based approach to generate bit-vector formulas representing instruction semantics. Rather than searching the functions representing instruction semantics in a large search space, they confine the search space by manually specifying semantics of basic instructions (called "base set" in [31]). Semantics of remaining instructions is then searched, starting from the semantics of basic instructions. For instance, using 6 manually provided templates, approach described in [25] is able to synthesize formulas for 534 32-bit x86 arithmetic instructions. Similarly, using 62 instructions in "base set", the approach proposed in [31] is able to synthesize SMT formulas for around 1795 of x86_64 instructions. Both the approaches have the advantages of not relying on the correctness of a code generator and that they can address the precision issue (semantics of EFLAGS) faced by EISSEC. We address the correctness problem by developing a systematic proof of soundness. Although both the approaches are effective in synthesizing the semantics of parts of 32-bit and 64-bit x86 instruction sets, they suffer from several limitations. First of all, the task of specifying the semantics of basic instructions demands reasonable understanding of the target architecture. Consequently, it is unclear the level of effort required in supporting new architectures. As compared to these approaches, EISSEC requires minimal (if not none) architecture knowledge (not the understanding of instruction semantics). Second, both the approaches could not extract some of the user-level instructions (especially, floating points and vector instructions in [25]) because of the difficulty or complexity modeling those instructions. In our experiments, EISSEC was able to extract semantics of advanced x86 instructions also. Moreover, the approach used in EISSEC is not limited by the size of instruction's input space (32-bit or 64-bit) and can easily scale to complex instruction sets.

7. CONCLUSION

In this paper, we described an automated approach using symbolic execution for extracting the instruction semantics model encoded in the code generators of modern compilers. We formulate the model extraction problem as the all paths problem, and employ carefully engineered symbolic execution system for extracting the complete input/output behavior of the code generators. Unlike existing symbolic execution systems that are mostly applied in the context of testing or bug finding, the model extraction problem demands different considerations in the design of a symbolic execution system.

Our experimental evaluation validates our hypothesis that the knowledge encoded in the code generators of modern compilers can be used for semantic extraction with relative ease. We apply EISSEC to GCC's x86 code generator (of size 120 KLoC) and extract semantics of all the instructions supported by the code generator. Importantly, we demonstrate that by exploiting the strength of symbolic execution in reaching all of the rarely visited corners of the program code, our approach is able to extract the semantics of all the instructions supported by the code generator. Furthermore, it reduces manual modeling efforts even further than existing learning based approaches. We also address a common concern of the soundness of the extracted model by developing a systematic proof. Lastly, we demonstrate architecture-neutrality of our approach by extracting semantic model of AVR architecture.

In the spirit of open-source community and to share our research prototype with other researchers working on similar problems, we provide artifact description of our approach in this paper. Additionally, the artifact is also available from our laboratory web site [28].

8. ARTIFACT DESCRIPTION

EISSEC artifact is packaged as a Docker image that can be installed on Linux, Windows, MacOS and many other OSes. Instructions for using the artifact are as follows:

- **Docker installation.** Please refer to steps at https://docs.docker.com/engine/installation/ to install Docker on your machine. EISSEC docker image has been tested on Ubuntu-14.04 x86_64.
- **Using EISSEC Docker image.** Execute commands below to pull EISSEC Docker image, create a container from the image and get a shell access to the container.

```
$ docker pull seclab/eissec
$ docker create -it --name eissec seclab/eissec
$ docker start eissec
$ docker exec -it eissec bash
```

```
_G2101 in 21..28,
_G653 in 36..138,
_G653 #>= _G588,
_G184 #>= _G653,
_G462 in -1..32,
_G462 + 10 #= _G494,
_G494 in 9..42,
_G494 + _G543 #= _G588,
_G462 + 10 #= _G494,
_G494 * 3 #= _G543,
_G543 in 27..126,
_G494 + _G543 #= _G588,
_G494 * 3 #= _G543,
_G588 in 36..137,
_G653 #>= _G588,
_G588 #=< _G184 + -1,
_G494 + _G543 #= _G588,
_G184 in 37..138,
_G184 #>= _G653, _G588 #=< _G184 + -1,

map:  cmpb    %_G2101 -> insn(_G9,
          rtl(15,_G27,union_u(
         [rtl(_G184,_G191,_G198,_G205),
          rtl(_G462,_G469,_G470,_G471),
          rtl(_G653,_G660,_G667,_G668)|_G393],
          _G39,_G46,_G53,_G54),_G73))
```

Figure 5: Mapping rule for `cmpb %reg` instruction

After executing these commands, we get shell access to the EISSEC container. Inside the container, in `eissec` directory, the layout of EISSEC source package can be seen. A `README` file under `eissec` directory explains the layout. Now execute commands below inside the container to build the EISSEC executable and extract the model from x86 code generator.

```
$ cd eissec
$ source env_setup.sh
$ make
$ cd test/x86
$ ./testmodel dummy.c
$ ./fullmodel dummy.c
```

- **Output of testmodel.**

  ```
  $ ./testmodel dummy.c
  ```

  This command will dump assembly to RTL mapping for a sample code generator. One of the dumped rules is shown in Fig. 5.

  In the Figure, `_G` are Prolog variables. The figure shows the mapping rule for `cmpb` x86 instruction. The operand of `cmpb` is represented by a variable `_G2101`. It is a register operand, as signified by the `%`-prefix before `_G2101`. RTL corresponding to the assembly specifies how it maps `_G2101` into RTL variables. For variables in assembly instruction, RTL variables are obtained by solving constraints involving RTL variables.

  Mapping rules are of the form $asm(x, y, z) = rtl$, where x, y, and z are variables in assembly and rtl is an RTL snippet. The RTL snippet contains many variables that are derived from the operands of the assembly instruction. Relations between x, y, z and rtl are captured by the set of constraints for every rule. There can be more than one possible instantiation that satisfies the constraints for a single rule. These instantiations correspond to different operands that a rule will support.

  Steps involved in extracting instruction semantics are:

  – *Transform GCC's code generator code for symbolic execution.*

    EISSEC uses CIL-based source-to-source transformation. Transformed code, when executed, performs symbolic execution of the code generator. Our source-to-source transformer is named `pkgs/cil-1.4.0/bin/cilly`. It transforms `insn-recog.c` (from `test/x86`) while compiling (`test/x86`). Transformed version of `insn-recog.c` can be found in `/tmp/insn-recog.cil.c`.

  – *Constraints sent to constraint solver.*

    When `testmodel` or `fullmodel` is executed, constraints sent to `cons_solve` are dumped into `/tmp/logcs` file. If you would like to change the location, please modify `driver.c` in `test/x86` and recompile.

- **Maturity.** Artifact describes EISSEC symbolic execution system used to extract Assembly-to-IR model from compiler's code generator. Extracting assembly-to-IR semantic model is the one of the complex tasks in the process of assembly-to-IR translation. Typical approach to this problem is to manually extract assembly-to-IR semantic model. EISSEC automates this task, and the *artifact demonstrates how EISSEC automatically extracts the model.*

  Note that this artifact does not describe *how to apply extracted semantic model for assembly-to-IR translation process.*

- **Source code.** EISSEC's source code is fully contained inside `eissec` directory inside the Docker container. The `src` directory inside `eissec` contains the source code of source-to-source transformer (`transformer`) (for symbolic execution), helper functions for symbolic execution (`symhelper`) and the interface to the constraint solver (`csolve`). Additionally, EISSEC source code can also be downloaded from http://seclab.cs.sunysb.edu/seclab/download.html.

9. REFERENCES

[1] Dagger. http://dagger.repzret.org.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[3] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from c protocol code by symbolic execution. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 331–340, 2011.

[4] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. Decompilation to compiler high ir in a binary rewriter. Technical report, Tech. rep., University of Maryland (November 2010), http://www. ece. umd. edu/barua/high-IR-technical-report10. pdf.

[5] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium*, Feburary 2011.

[6] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *Compiler Construction*. Springer, 2005.

[7] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Conference on Annual Technical Conference*, 2005.

[8] Derek L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Cambridge, MA, USA, 2004.

[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: a binary analysis platform. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, 2011.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008.

[11] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International Conference on Model Checking Software*, SPIN'05, pages 2–23, 2005.

[12] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 322–335, 2006.

[13] Vitaly Chipounov and George Candea. Dynamically Translating x86 to LLVM using QEMU. Technical report, 2010.

[14] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[15] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout - a retargetable dynamic binary translation framework. In *In Proceedings of the 2002 Workshop on Binary Translation*, 2002.

[16] Cristina Cifuentes, Mike Van Emmerik, and Norman Ramsey. The design of a resourceable and retargetable binary translator. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 280–291. IEEE, 1999.

[17] Jack W. Davidson and Christopher W. Fraser. Code Selection Through Object Code Optimization. *ACM Trans. Program. Lang. Syst.*, 1984.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.

[19] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of the USENIX Conference on Annual Technical Conference*, 2007.

[20] Ulfar Erlingsson, Silicon Valley, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: software guards for system address spaces. In *OSDI*, 2006.

[21] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, 2007.

[22] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 47–54, 2007.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, 2005.

[24] Patrice Godefroid, Michael Y Levin, and David A Molnar. Automated Whitebox Fuzz Testing. In *Network Distributed Security Symposium (NDSS)*, volume 8, pages 151–166, 2008.

[25] Patrice Godefroid and Ankur Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[26] Niranjan Hasabnis. *Automatic Synthesis of Instruction Set Semantics and its Applications*. PhD thesis, Stony Brook, NY, USA, August, 2015.

[27] Niranjan Hasabnis, Rui Qiao, and R. Sekar. Checking Correctness of Code Generator Architecture Specifications. In *International Symposium on Code Generation and Optimization*, CGO, 2015.

[28] Niranjan Hasabnis and R Sekar. EISSEC - Extracting Instruction Semantics by Symbolic Execution of Code Generators - software release. http://seclab.cs.sunysb.edu/seclab/eissec/.

[29] Niranjan Hasabnis and R. Sekar. Automatic Generation of Assembly to IR Translators Using Compilers (short paper). In *Workshop on Architectural and Microarchitectural Support for Binary Translation (in conjuction with CGO)*, AMAS-BT, 2015.

[30] Niranjan Hasabnis and R. Sekar. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[31] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, 2016.

[32] J.N. Hooker. Solving the incremental satisfiability problem. *The Journal of Logic Programming*, 15(1–2):177 – 186, 1993.

[33] Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. Disirer: Converting a retargetable compiler into a multiplatform binary translator. *ACM Trans. Archit. Code Optim.*, 7(4):18:1–18:36, December 2010.

[34] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security Symposium*, 2002.

[35] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. KLOVER: a symbolic execution and automatic test generation tool for C++ programs. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 609–615, 2011.

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace,

Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, 2005.

[37] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, 2002.

[38] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07. ACM, 2007.

[39] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network Distributed Security Symposium (NDSS)*, 2005.

[40] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, 2010.

[41] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *The Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[42] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, 2008.

[43] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010.

[44] R Sekar, IV Ramakrishnan, and Andrei Voronkov. Term indexing, handbook of automated reasoning, 2001.

[45] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive Pattern Matching. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, ICALP '92, 1992.

[46] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, 2005.

[47] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, December 2008.

[48] Markus Triska. The finite domain constraint solver of SWI-Prolog. In *FLOPS*, volume 7294 of *LNCS*, pages 307–316, 2012.

[49] Markus Triska. *Correctness Considerations in CLP(FD) Systems*. PhD thesis, Vienna University of Technology, 2014.

[50] Shaohui Wang, Srinivasan Dwarakanathan, Oleg Sokolsky, and Insup Lee. High-level model extraction via symbolic execution. Technical Report MS-CIS-12-04, Department of Computer and Information Science, University of Pennsylvania, 2012.

[51] J. Whittemore, Joonyoung Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Design Automation Conference, 2001. Proceedings*, pages 542–545, 2001.

[52] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[53] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[54] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[55] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Stephen McCamant, Laszlo Szekeres, Dawn Song, and Wei Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[56] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2014.

[57] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.