Language-Parametric Compiler Validation with Application to LLVM

Theodoros Kasampalis
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
kasampa2@illinois.edu

Daejun Park
Runtime Verifcaction, Inc.
Urbana, Illinois, USA
daejun.park@runtimeverification.com

Zhengyao Lin
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
zl38@illinois.edu

Vikram S. Adve
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
vadve@illinois.edu

Grigore Roșu
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
grosu@illinois.edu

ABSTRACT

We propose a new design for a Translation Validation (TV) system geared towards practical use with modern optimizing compilers, such as LLVM. Unlike existing TV systems, which are custom-tailored for a particular sequence of transformations and a specific, common language for input and output programs, our design clearly separates the transformation-specific components from the rest of the system, and generalizes the transformation-independent components. Specifically, we present Keq, the first program equivalence checker that is parametric to the input and output language semantics and has no dependence on the transformation between the input and output programs. The Keq algorithm is based on a rigorous formalization, namely cut-bisimulation, and is proven correct. We have prototyped a TV system for the Instruction Selection pass of LLVM, being able to automatically prove equivalence for translations from LLVM IR to the MachineIR used in compiling to x86-64. This transformation uses different input and output languages, and as such has not been previously addressed by the state of the art. An experimental evaluation shows that Keq successfully proves correct the translation of over 90% of 4732 supported functions in GCC from SPEC 2006.

CCS CONCEPTS

• **Software and its engineering → Compilers**; **Software verification and validation**.

KEYWORDS

Translation Validation, Compilers, Program Equivalence, Simulation

ASPLOS '21, April 19–23, 2021, Virtual, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446751

ACM Reference Format:
Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roșu. 2021. Language-Parametric Compiler Validation with Application to LLVM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3445814.3446751

1 INTRODUCTION

Modern optimizing compilers such as LLVM [19] and GCC [11] have evolved into intricate systems with huge code bases and, consequently, uncaught bugs that make it into mature releases [38]. The reality of compilation bugs combined with lack of any formal correctness guarantee for the compilation process harms software development and limits the guarantees other software systems can provide. For example, most iOS applications and all of the watchOS and tvOS applications are shipped by developers to the Apple Store as LLVM bitcode [7, 14] and they are compiled to machine code on Apple's servers, thus allowing the possibility of unintended behaviors introduced to the binary due to compilation errors [42]. More generally, other systems that aim for formal guarantees of correctness, such as sel4 [35], also depend on provable correctness of translations from source to binary code. These circumstances have motivated broad interest in *compilation verification*: providing a formal guarantee that a compilation of a program is correct. In this work, we attack the problem of compilation verification not only theoretically, as an instance of program equivalence, but from a practical standpoint as well: we are after a solution appropriate for real-world optimizing compilers.

Translation Validation (TV) [30] is a compilation verification technique that aims to prove correctness of a single instance of compilation, by considering only the specific input and output programs. TV techniques are well-suited to the compilation verification problem because they can be *composed* to validate a sequence of compilation steps, they can be *retrofitted* to existing production compilers, and they can be maintained independently from the compiler itself.

The basic components of a TV system are as follows:

- A formal notion of program equivalence.

- A verification condition (VC) generator that generates a sufficient set of obligations to be discharged to prove equivalence.
- A proof system that accepts the verification conditions, generates a machine-checkable equivalence proof, and checks the proof for correctness.

There is a rich literature of successful TV systems for compilation verification (see Section 6). The main limitation of these systems is that each of them is custom-tailored for a particular sequence of transformations and a specific, common intermediate language for input and output programs. For example, Necula's work on GCC [29] is limited to the Register Transfer Language (RTL), and does not apply to the transformations on the higher-level GIMPLE representation [10]. Moreover, none of these previous systems would be able to directly verify a key phase such as Instruction Selection in LLVM, which converts between two different IRs. The best effort has been to translate both input and output programs to a third, common internal representation as a preliminary step [35], which introduces two new unverified language translators in order to verify the original translator.

In this work, we present a TV system that consists of modular components designed to be independent of the various transformations and languages found in compilers. Specifically, the key insight underlying our work is that two of the three TV system components mentioned above can be generalized to be *transformation- and language-independent*: the formal notion of equivalence, and the proof system.

For our **proof system**, we design a program equivalence checker, Keq, that does not depend on the transformation pass at hand and the input/output language pair. Keq takes as input the operational semantics of the input and output languages, as well as the VC for a transformation sequence. The operational semantics of each language must be defined once, and then can be used with Keq for any transformation that involves these languages. The VC used by Keq is a set of pairs of relevant program states, which we call *synchronization points*, Moreover, the input and output languages can be completely different, as long as programs can be related using the VC. In this work, we showcase the power of these properties by using Keq in a prototype TV system for the Instruction Selection phase of LLVM, a sophisticated phase that translates LLVM IR [22] to Machine IR [24] representing the x86-64 instruction set. Moreover, in our ongoing work (not part of this paper), we are applying Keq *unchanged* to validate the register allocation phase of LLVM, with a VC generator that treats the allocator completely as a black box (i.e., has no knowledge of the allocation algorithm), and we plan to apply Keq to LLVM-to-LLVM transformations in future.

We formalize the Keq algorithm and prove it correct by defining a **formal notion of equivalence** that enables a language-parametric proof system. In particular, we present a new formalization of program equivalence, namely *cut-bisimulation*, that generalizes different weak bisimulation variants that have been used in existing TV systems. An equivalence proof in Keq involves proving that a given VC is a cut-bisimulation for the input and output programs. As we will discuss in Section 2, cut-bisimulation is better-suited for program equivalence proofs because it offers a formal way to address the need for flexible synchronization points and, when needed, to abstract away the complicated correspondence between program states in different languages.

```
unsigned arithm_seq_sum(unsigned a0, unsigned d,
                        unsigned n) {
  unsigned s = a0, a = a0, i;
  for (i = 1; i < n; ++i) {
    a = a + d;
    s = s + a;
  }
  return s;
}
```

Figure 1: Function to compute the sum of the first n elements of an arithmetic sequence with first element a0 and step d.

As an example, consider the simple C code shown in Figure 1. Figure 2 shows the mid-level internal representation (IR) of this code in the LLVM Compiler Infrastructure (the LLVM IR, or simply, LLVM), as well as the output of the instruction selection (ISel) phase of the compiler when compiling for x86-64. This phase is the primary language translation step beyond the front-end: it translates LLVM IR to a low-level IR called Machine IR representing a particular target instruction set (ISA). The Machine IR for x86-64 keeps some high-level abstractions such as an unlimited number of virtual registers and support for SSA virtual registers, along with the x86-64 ISA opcodes; we call this output language "Virtual x86". In Figure 3, points $\{p0, p1, p2, p3\}$ are corresponding synchronization points where state comparisons are valid for live values between the LLVM IR and the machine code generated by the ISel phase. Using these points and appropriate LLVM IR and Virtual x86 semantics definitions, Keq proves that the synchronization point relation is a cut-bisimulation and hence the two programs are equivalent (see Section 3 for more details).

Finally, the **verification condition generator** has to take into account the specifics of the transformation in question (either using compiler-generated hints or heuristics inspired by the transformation's effect in input programs). For this reason, it is not clear how to effectively generalize it, although there have been examples in prior work of verification condition generators able to work with a wider range of transformations (see Section 6).

We have implemented Keq as a tool within the $\mathbb{K}$ framework [32]. Keq accepts $\mathbb{K}$ formal operational semantic definitions for the input and output languages. In order to use Keq for TV of Instruction Selection in LLVM, we have developed $\mathbb{K}$ semantic definitions of a subset of the LLVM and Virtual x86 languages. To generate synchronization points for Instruction Selection, we have also developed a verification condition generator as a python script that relies on a minimal hint generator added to the LLVM compiler. The hint generation code that we needed to add to LLVM contains less than 500 lines of C++. For comparison, the Instruction Selection pass implementation uses more than 140,000 lines of code. All the above constitute a prototype TV system for the Instruction Selection pass of the LLVM compiler infrastructure.

We evaluate our prototype on 4732 functions of the GCC SPEC 2006 benchmark that are covered by the fragment of LLVM and x86 language semantics we developed. We correctly validate the

```
define i32 @arithm_seq_sum(i32 %a0, i32 %d, i32 %n) {
entry:      ; p0
  br label %for.cond

for.cond:    ; p1, p2
  %s.0 = phi i32 [ %a0, %entry ], [ %add1, %for.inc ]
  %a.0 = phi i32 [ %a0, %entry ], [ %add, %for.inc ]
  %i.0 = phi i32 [ 1, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i32 %i.0, %n
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %add = add i32 %a.0, %d
  %add1 = add i32 %s.0, %add
  br label %for.inc

for.inc:
  %inc = add i32 %i.0, 1
  br label %for.cond

for.end:  ; p3
  ret i32 %s.0
}
```

(a) *LLVM IR*

```
arithm_seq_sum:
.LBB0:     ; p0
  %vr8_32  = COPY edx
  %vr7_32  = COPY esi
  %vr6_32  = COPY edi
  %vr9_32  = mov  1
  jmp  .LBB1
.LBB1:     ; p1, p2
  %vr0_32  = PHI  %vr6_32, .LBB0, %vr4_32, .LBB3
  %vr1_32  = PHI  %vr6_32, .LBB0, %vr3_32, .LBB3
  %vr2_32  = PHI  %vr9_32, .LBB0, %vr5_32, .LBB3
  %vr10_32 = sub  %vr2_32, %vr8_32
  jae  .LBB4
  jmp  .LBB2
.LBB2:
  %vr3_32  = add  %vr1_32, %vr7_32
  %vr4_32  = add  %vr0_32, %vr3_32
  jmp  .LBB3
.LBB3:
  %vr5_32  = inc  %vr2_32
  jmp .LBB1
.LBB4:
  eax      = COPY %vr0_32
  ; p3
  ret
```

(b) *Virtual x86*

Figure 2: The arithmetic sequence sum in LLVM IR and Virtual x86, as produced by Instruction Selection at optimization level O0. Comments in red show the synchronization points generated by our prototype.

Sync Point	*Prev BB* (LLVM)	*Prev BB* (Vx86)	*Equality Constraints*	
*p*0	-	-	%a.0 = **edi**, %n = **edx**,	%d = **esi**,
*p*1	%entry	.LBB0	%d = %vr7_32, %a.0 = %vr6_32,	1 = %vr9_32, %n = %vr8_32,
*p*2	%for.**inc**	.LBB3	%add = %vr3_32, %add1 = %vr5_32, %inc = %vr5_32	%n = %vr8_32, %d = %vr7_32,
*p*3 (exit)	-	-	%s.0 = **eax**	

Figure 3: Synchronization points for the translation of the arithmetic sequence sum in Figure 2. A more detailed explanation will be given in Section 3.

translation of 91.52% of the supported functions in GCC, i.e., 4331 / 4732 functions. Additionally, we reintroduce two real Instruction Selection bugs to the code base and show that the buggy compilations could *not* pass our system.

In short, this work presents the first TV system that clearly separates the transformation-specific components (i.e., VC generators) from the transformation-independent ones (i.e., proof system and language semantics), and generalizes the latter to be parameterized to different languages. This way, our design reduces the amount of work needed per transformation and (intermediate) language: a semantic definition of every language found in the compilation path and one or more verification condition generators that use transformation-specific information. In summary, the main contributions of this paper are:

- KEQ, a new tool for checking program equivalence that accepts the operational semantics of the input and output languages as parameters, and is independent of the transformation used to generate the output. This is the first program equivalence checking tool known to the authors that is language-parametric instead of containing hard-coded language semantics as is the norm in the literature.
- A new rigorous formalization of program equivalence, called *cut-bisimulation*, that generalizes weak bisimulation variants that have been traditionally used in different TV systems. We use cut-bisimulation as the basis of the KEQ equivalence checking algorithm, and provide a correctness proof for that algorithm.
- A prototype of a Translation Validation system for the Instruction Selection pass of the LLVM compiler infrastructure, able to automatically prove equivalence for translations from LLVM IR when compiling to the x86-64 instruction set. This is a mature, sophisticated translation phase of a production compiler. Moreover this transformation uses different input

and output languages, and as such has not been previously addressed by the state of the art.

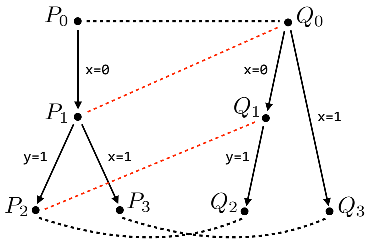
2 FORMALIZING PROGRAM EQUIVALENCE

Intuitively, two (possibly non-terminating) programs are "equivalent" when, given the same input, they reach the same relevant states in effectively the same order. Similarly, a program A "refines" another program B when A contains no new behaviors that do not exist in B. Strong bisimulation [34] is a formal way of expressing this requirement: two programs are bisimilar, and hence equivalent[1], when they reach only equivalent states throughout their execution. Equivalence proofs based on strong bisimulation then require the existence of a relation of equivalent states that covers all possible states of both programs. This is too strong of a requirement for practical program equivalence proof systems.

Consider the simple program transformation example shown in Figure 4(a), commonly performed by compilers as part of partial redundancy elimination. The seemingly equivalent two programs are still not strongly bisimilar, mainly because the intermediate states (P_1 and Q_1) are not "similar". Weaker variants, such as stuttering or branching bisimulation [34], could be used to prove their equivalence, since they are flexible to admit the irrelevant intermediate states. Figure 4(b) depicts a stuttering bisimulation relation shown as both black and red dotted lines, where the transitions $P_0 \longrightarrow P_1$ and $Q_1 \longrightarrow Q_2$ are considered "stuttering" transitions.[2] Note that, however, identifying the stuttering transitions are non-trivial. Indeed, the irrelevant intermediate states P_1 and Q_1 have the potential to stutter with all adjacent states. As such, there exist many candidate stuttering transitions (which are $P_0 \longrightarrow P_1, P_1 \longrightarrow P_2$, $P_1 \longrightarrow P_3$, $Q_0 \longrightarrow Q_1$, and $Q_1 \longrightarrow Q_2$ in this example) and identifying the appropriate ones among many candidates is not straightforward. The problem of identifying stuttering transitions becomes apparent when we consider the witness-based translation validation approach [28], in which the candidate relation is generated by the compiler as a "witness" for the correctness of the transformation, and proving the equivalence is reduced to checking that the generated relation is a (bi)simulation. However, it is not easy for the compiler to identify stuttering transitions which are not directly related to the internal information used in the compiler transformation. Thus, the stuttering transitions should be inferred separately, which incurs additional overhead in proving equivalence.[3]

```
P:  x = 0; if (*) { y = 1; } else { x = 1; }
Q:  if (*) { x = 0; y = 1; } else { x = 1; }
```

(a)



(b)

Figure 4: Program transformation example (as part of partial redundancy elimination), a stuttering bisimulation relation (both black and red dotted lines), and a cut-bisimulation relation (only black dotted lines). The `if(*)` statement denotes the non-deterministic branching operation.

Ideally, we want a bisimulation variant that allows us to relate only the relevant states at which the two programs actually match each other, e.g., at the start of corresponding functions or basic blocks, at the loop headers, etc., without ever considering irrelevant states at all. We also want to be able to control the granularity of these points depending on the transformation at hand. We call such pairs of relevant states *synchronization points*. For each program, we call the set of states related through a synchronization point with a state of the other program a *cut*. The intuition for the cut of a program is that the states in the cut suffice as observation points of the program behavior, that is, nothing relevant can happen which is not witnessed by a cut state. Then we can define bisimulations only between cut states; we call these *cut-bisimulations*. For the example in Figure 4, simply the synchronization relations, indicated by only black dotted lines, define a cut-bisimulation relation and thus prove equivalence.

In order for cut-bisimulations to correctly capture program equivalence, two conditions must be satisfied. First, there must be enough cut states for the two programs so that no relevant behavior of one program can pass unsynchronized with a behavior of the other program. This implies, in particular, that each final state must be in the cut. It also implies that each infinite execution must contain infinitely many cut states, because otherwise one of the programs may not terminate while the other terminates.

Second, any two states related by a cut-bisimulation must be compatible. For example, if we want to show that the LLVM and Virtual x86 programs in Figure 2 are equivalent, then the two final states at synchronization point $p3$ must satisfy the condition that the values held by the LLVM local variable `%s.0` and by the x86-64 register `eax` are "the same". What it precisely means for two values in different languages to be the same is not trivial, due to different representations (e.g., big-endian vs little-endian), different memory layouts (physically same location may point to different values, or contain garbage that has not been collected yet), etc. Also, state compatibility may require to check if environment variables, input/output buffers, etc., are also "the same". Moreover, states corresponding to undefined behaviors (e.g., division by zero) may or may not be desired to be compatible, e.g., when compiler optimizations that may improve performance by assume such behaviors are illegal and so cannot occur. We found it awkward to encode such complex state compatibility abstractions as labels on transitions, as the existing notions of bisimulation require. Instead, we parameterize our theoretical results, algorithms and implementation with a binary relation on states $\mathcal{A}$, which we call an *acceptability* (or *compatibility* or *indistinguishability*) relation.

[1] For simplicity, we say "program equivalence" and "bisimulation", but our results and algorithms also support "program refinement" and "simulation".

[2] More precisely, it is a stuttering bisimulation over the Kripke structure where the labeling function L satisfies $L(P_0) = L(P_1)$ and $L(Q_1) = L(Q_2)$.

[3] The time complexity of the best known algorithm for inferring stuttering bisimulation is $O(m \log n)$ where m is the number of transitions and n is the number of states [12].

We next give intuitive but informal definitions for *cut* and *cut-bisimilarity*, as well as the sketch for a theorem that allows us to use cut-bisimilarity for program equivalence. Section 7 provides a rigorous formalization for these definitions and theorems as well as formal proofs.

Definition 2.1 (Cut). *Given a program P, a set of states C is a cut for P if (a) the start state belongs to C (i.e., is a cut state), (b) any terminating execution of P terminates in C, and (c) any non-terminating execution of P goes through C infinitely often, in every finite number of steps.*

Definition 2.2 (Cut-Bisimilarity). *A relation between states of two programs is a cut-bisimulation if and only if (a) all related states are cut states, and (b) starting from a pair of related states, the programs always reach another pair of related cut states by going through only non-cut states. If such relation exists for two prorgams, they are cut-bisimilar.*

Theorem 2.3 (Cut-Bisimilarity and Program Equivalence). *Two cut-bisimilar programs, where all the related (cut) states in the corresponding cut-bisimulation are also related in the acceptablility relation, are equivalent.*

3 LANGUAGE-INDEPENDENT EQUIVALENCE CHECKING ALGORITHM AND IMPLEMENTATION IN $\mathbb{K}$

We implemented a language-independent equivalence checking tool on top of the $\mathbb{K}$ framework [32]. $\mathbb{K}$ provides a language for defining operational semantics of programming languages, and a series of generic tools that take one or more language semantics as input and operate on programs in those languages. We developed a new tool, Keq, which takes as input two language semantics and two programs, one in each language, along with a (symbolic) synchronization relation, and checks whether the two programs are indeed equivalent with the synchronization relation as witness.

Note that checking program equivalence in Turing-complete languages is equivalent to checking the totality of a Turing machine (whether it terminates on all inputs), which is undecidable [31]. The best we can do is to find techniques and algorithms that work well enough in practice. Theoroem 2.3 suggests such a technique: find a (witness) relation P and show that it is a cut-bisimulation. While finding such a relation is hard in general, it is easier to check if a given relation, for example one produced by an instrumented compiler (see Section 4.5), is a cut-bisimulation.

Our Keq implementation follows the model of the theoretical Algorithm 1. Function main essentially checks whether P is a cut-bisimulation: for each pair $(p_1, p_2) \in \mathsf{P}$, if $p_1 \rightsquigarrow_1 p_1'$ for some p_1', then there should exist p_2' with $p_2 \rightsquigarrow_2 p_2'$ such that $p_1' \ \mathsf{P}\ p_2'$; and the converse. It first gets the cut-successors of p_i (at line 7), and checks whether each pair of the successors is related in P (line 9). The pairs found to be related in P are marked in black (line 10), while the others remain in red. If all of the successors are in black, it returns true (line 12). Note that the algorithm can also be used for checking whether P is a cut-simulation, by simply considering only N_1 in the line 12, i.e., replacing the if-condition with $\forall n \in N_1.\ n.\text{color} = \text{black}$. The correctness proof of Algorithm 1 is provided in Section 8.

```
Data: $T_i = (S_i, \xi_i, \to_i, C_i)$; $\mathsf{P} \subseteq C_1 \times C_2$;
1  Function main():
2      foreach $\boxed{(p_1, p_2) \in \mathsf{P}}$ do                // $\boxed{\overline{\mathsf{P}}}$
3          if check($p_1, p_2$) = false then
4              return false;
5      return true;
6  Function check($p_1, p_2$):
7      $N_1 \leftarrow$ next$_1$($p_1$);  $N_2 \leftarrow$ next$_2$($p_2$);
8      foreach $(n_1, n_2) \in N_1 \times N_2$ do
9          if $\boxed{(n_1, n_2) \in \mathsf{P}}$ then            // $\boxed{[\![(n_1, n_2)]\!] \subseteq [\![\overline{\mathsf{P}}]\!]}$
10             $n_1$.color ← black;  $n_2$.color ← black;
11     if $\forall n \in N_1 \cup N_2.\ n$.color = black then
12         return true;
13     return false
14 // Returns cut-successors of $n$
15 Function next$_i$($n$):
16     $N \leftarrow \{n\}$;  $Ret \leftarrow \emptyset$;
17     while $N$ is not empty do
18         choose $n$ from $N$;  $N \leftarrow N \setminus \{n\}$;
19         $N' \leftarrow \{n' \mid n \boxed{\to_i} n'\}$;           // $\boxed{\overline{\to_i}}$
20         foreach $n' \in N'$ do
21             if $\boxed{n' \in C_i}$ then                     // $\boxed{[\![n']\!] \subseteq [\![\overline{C_i}]\!]}$
22                 $n'$.color ← red;
23                 $Ret \leftarrow Ret \cup \{n'\}$;
24             else
25                 $N \leftarrow N \cup \{n'\}$;
26     return $Ret$;
```

Algorithm 1: Equivalence checking algorithm. For checking cut-simulation, replace $N_1 \cup N_2$ with N_1 at line 11. As given, the algorithm works with concrete data and thus is not practical. Replace boxed expressions with their grayed variants to the right for a practical, symbolic algorithm, as implemented in Keq.

Symbolic Implementation of Algorithm 1. Due to its concrete (as opposed to symbolic) nature, Algorithm 1 may not terminate when P is infinite. For example, P may include all the synchronization points at the beginning of the main loop in a reactive system implementation. Nevertheless, in practice it is often the case that we can over-approximate infinite sets symbolically. For example, we can use a logical formula φ to describe a symbolic state, which denotes a potentially infinite set $[\![\varphi]\!]$ of concrete states that satisfy it. Then we may be able to describe the sets of states S_i and C_i of the cut transition systems T_i $(i \in \{1, 2\})$ with finite sets $\overline{S_i}$ and $\overline{C_i}$, respectively, of symbolic states. Similarly, symbolic pair (φ, φ') can describe infinite sets $[\![(\varphi, \varphi')]\!]$ of pairs of states in the two transition systems, related through free/symbolic variables that φ and φ' can share. Then we may also be able to describe P as a finite set $\overline{\mathsf{P}}$ of pairs of symbolic states. If all these are possible, then Algorithm 1 can be modified by replacing the boxed expressions with their symbolic variants (in grey boxes); and n, n', n_1, n_2, p_1, p_2, etc., are symbolic now.

Given an operational semantics of a programming language, $\mathbb{K}$ provides us with an API to calculate symbolic successors of symbolic program configurations. This allows us to conveniently implement the symbolic $\overline{\longrightarrow_i}$ transitions. Also, $\mathbb{K}$ is fully integrated with the Z3 solver [9], allowing us to implement the set inclusion checks, i.e., $[\![(n_1, n_2)]\!] \subseteq [\![\overline{P}]\!]$ (at line 9) and $[\![n']\!] \subseteq [\![\overline{C_i}]\!]$ (at line 21), by requesting Z3 to solve the implications of the corresponding formulae. (See below.)

Optimizing SMT Queries. Before checking the symbolic set inclusion $[\![(n_1, n_2)]\!] \subseteq [\![\overline{P}]\!]$, we check first the equivalence between the path conditions of the two symbolic states n_1 and n_2, since the SMT query for checking the set inclusion becomes much simpler when the two path conditions are equivalent.[4] Let the path conditions of n_1 and n_2 be φ_1 and φ_2, respectively. Then, we need to prove $\varphi_1 \Rightarrow \varphi_2$ and $\varphi_2 \Rightarrow \varphi_1$ to prove the path condition equivalence. For proving $\varphi_1 \Rightarrow \varphi_2$, we ask Z3 to prove that its negation is unsatisfiable, that is, that $\varphi_1 \wedge \neg\varphi_2$ is unsatisfiable. (Similarly for proving $\varphi_2 \Rightarrow \varphi_1$ as well.) However, we found that Z3 performs poorly for proving the unsatisfiability of $\varphi_1 \wedge \neg\varphi_2$, especially because of the negation applied to φ_2 that involves existential quantifiers.

To improve the performance of Z3 solving, we devised the following optimization that is applicable when the underlying transition systems are deterministic. Suppose that $N_1 = \{n_1, \cdots\}$ and $N_2 = \{n_2, n_2', n_2'', \cdots\}$ (at line 7). Let the path condition of $(n_1, \cdots)$ and $(n_2, n_2', n_2'', \cdots)$ be $(\varphi_1, \cdots)$ and $(\varphi_2, \varphi_2', \varphi_2'', \cdots)$, respectively. Since the transition systems are deterministic, we have that $(\varphi_2 \vee \varphi_2' \vee \varphi_2'' \vee \cdots)$ is a tautology and φ_2 is disjoint from $\Psi_2 = (\varphi_2' \vee \varphi_2'' \vee \cdots)$. Thus, $\varphi_1 \wedge \neg\varphi_2$ is equivalent to $\varphi_1 \wedge \Psi_2$. Now, for proving $\varphi_1 \Rightarrow \varphi_2$, we ask Z3 to prove the unsatisfiability of $\varphi_1 \wedge \Psi_2$ (instead of $\varphi_1 \wedge \neg\varphi_2$). Note that Z3 performs much better for solving the positive form $\varphi_1 \wedge \Psi_2$ than the original negative form $\varphi_1 \wedge \neg\varphi_2$, even though the two are logically equivalent in theory, and the positive form is larger in size than the negative form.

This optimization has been adopted in our TV prototype in Section 4, leveraging the fact that both our source and target language semantics (LLVM and x86) are deterministic.

Example. We implemented the symbolic variant of Algorithm 1 in a tool called Keq for checking language-independent program equivalence.[5] To illustrate how Keq works, consider the running example in Figure 2. At the beginning of the programs, we have the symbolic synchronization point $p0$ which is a triple $(s_{p0}, s'_{p0}, \psi_{p0})$, where

$$s_{p0} \equiv \texttt{\%a0} \mapsto a_0 * \texttt{\%d} \mapsto d_0 * \texttt{\%n} \mapsto n_0$$
$$s'_{p0} \equiv \texttt{edi} \mapsto a_0' * \texttt{esi} \mapsto d_0' * \texttt{edx} \mapsto n_0'$$
$$\psi_{p0} \equiv a_0 = a_0' \;\wedge\; d_0 = d_0' \;\wedge\; n_0 = n_0'$$

are the symbolic state of the LLVM program, the symbolic state of the x86 program ($*$ is a separator for map bindings), and the constraint for s_{p0} and s'_{p0} to be related, essentially saying that the inputs of the two programs are the same. Mathematically, $p0$ denotes the set of infinitely many pairs of states $\{(s_{p0}, s'_{p0}) \mid \psi_{p0}\} = \{(\texttt{\%a0} \mapsto a * \texttt{\%d} \mapsto d * \texttt{\%n} \mapsto n,\ \texttt{edi} \mapsto a * \texttt{esi} \mapsto d * \texttt{edx} \mapsto n) \mid a, d, n \in \mathbb{N}\}$ (an over-approximation including all the pairs of interest). Symbolic synchronization points $p1$, $p2$, and $p3$ are similarly defined (see Figure 3).

Next we illustrate how Keq symbolically runs Algorithm 1. Let $\mathsf{P} = \{p0, p1, p2, p3\}$. First, Keq picks one point (say $p0$) from P (line 2 of Algorithm 1) and executes the function check with it. In check, it first symbolically executes each program (lines 7 and 19) until they reach another synchronization point (line 21). In our case they reach $p1$ with the pair of symbolic states:

$$s_{p1} \equiv s_{p0}$$
$$s'_{p1} \equiv s'_{p0} * \texttt{\%vr8} \mapsto n_0' * \texttt{\%vr7} \mapsto d_0' * \texttt{\%vr6} \mapsto a_0' * \texttt{\%vr9} \mapsto 1$$

Keq checks if $\{(s_{p1}, s'_{p1}) \mid \psi_{p0}\}$ is included in $p1$ (line 9), which is true.

Next, suppose Keq picks $p2$ (line 2). Symbolic execution starting from $p2$ yields two pairs of symbolic traces, that reach synchronization points $p2$ (through the for-loop body) and $p3$ (escaping the for-loop), respectively. Let us consider the first case. We have the pair of the final states:

$$\begin{aligned} s_{p2} \equiv\ & (\texttt{\%d} \mapsto d_2 * \texttt{add1} \mapsto s_2 + a_2 + d_2 * \texttt{\%n} \mapsto n_2 \\ & * \texttt{add} \mapsto a_2 + d_2 * \texttt{inc} \mapsto i_2 + 1) \;\wedge\; (i_2 < n_2) \\ s'_{p2} \equiv\ & (\texttt{\%vr7} \mapsto d_2' * \texttt{\%vr4} \mapsto s_2' + a_2' + d_2' * \texttt{\%vr8} \mapsto n_2' \\ & * \texttt{\%vr3} \mapsto a_2' + d_2' * \texttt{\%vr5} \mapsto i_2' + 1) \;\wedge\; (i_2' - n_2' < 0) \\ \psi_{p2} \equiv\ & a_2 = a_2' \wedge d_2 = d_2' \wedge s_2 = s_2' \wedge i_2 = i_2' \wedge n_2 = n_2' \end{aligned}$$

Keq checks if $\{(s_{p2}, s'_{p2}) \mid \psi_{p2}\}$ is included in $p2$ (line 9), which is true. Regarding the path conditions, Keq checks if $i_2 < n_2$ and $i_2' - n_2' < 0$ are equivalent given ψ_{p2}, which is true.

The other case for $p3$ is similar and check with $p2$ returns true. Then, Keq continues to pick from the remaining synchronization points and execute check with each of them (loop at lines 2-4), eventually returning true (line 5).

4 TRANSLATION VALIDATION FOR LLVM INSTRUCTION SELECTION

Here we describe the application of the proposed equivalence checking algorithm in a translation validation system for the Instruction Selection phase of LLVM. This phase translates the LLVM intermediate representation (IR) into various target instruction sets, and we focus on the x86-64 target for the scope of this work. We chose this particular application because it is a non-trivial component of a widely-used mature compiler that operates with different input and output languages. Moreover, in an LLVM-based compiler (e.g., Clang [6], Swift [39], Julia [16]), this phase is the primary language translation step beyond the front-end: it converts the mid-level IR to the low-level Machine IR. As explained in Section 6, none of the existing TV techniques can be directly used to validate instruction selection, because it translates between two fundamentally different languages.

The various components of the system along with the Instruction Selection (ISel) phase itself are shown in Figure 5. ISel translates

[4] In case that the two path conditions are not equivalent, we can split the symbolic states with different path conditions and re-run the loop (lines 8–10).

[5] Keq also supports program refinement, but for simplicity we only discuss equivalence.

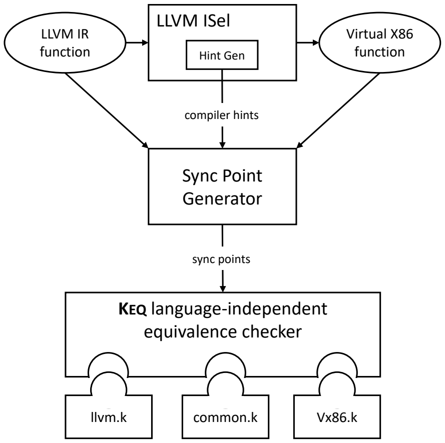


Figure 5: Translation validation system for LLVM ISel phase

LLVM IR to Machine IR; when targeting x86-64, the generated machine IR represents a slightly simplified version of the x86-64 instruction set that we call "Virtual x86". Let us discuss the various components of the TV system in more detail.

Verification Condition Generator. We enhanced ISel with a hint generator to output information relevant to the specific translation instance. This information along with the input and output programs guides the generation of the synchronization points for the translation instance. These two components, the hint generator and the synchronization points generator, constitute the Verification Condition (VC) generator. A key observation is that the hint generator will be part of the compiler code base and maintained by compiler engineers, so we want its implementation to be possible without any formal methods expertise.

The specific strategy for synchronization point generation employed by our VC generator for ISel is described in more detail in Subsection 4.5. In general, the objective of a VC generator for use with Keq is to provide a set of points that form a cut for the input and output programs and are adequate for a cut-bisimulation proof by Keq. Determining a strategy for the generation of such points requires understanding of the target transformation's assumptions and effects on the code, as well as formal methods knowledge about bisimulation proofs.

Typically, a successful VC generator will need some information about the effects of the target transformation on the code. For example, our VC generator requires knowledge of the mapping from LLVM IR virtual registers to Virtual x86 virtual and/or physical registers that was used during the translation of the target input program. Such information can be either automatically obtained by an appropriate inference algorithm or provided by enhancing the compiler with a hint generator. The former approach treats the compiler as a completely black box, while the latter trades off transparency for increased accuracy. In case of a hint generator, we emphasize that this specific component should not require any formal methods expertise to be developed or maintained.

Language Semantics. The set of synchronization points is provided to Keq, which is parameterized by the $\mathbb{K}$ semantic definitions of LLVM IR and Virtual x86. These are discussed in more detail in Subsections 4.2 and 4.3. In general, one needs a $\mathbb{K}$ semantic definition for the input and output languages involved in the target transformation (which may be identical if the language is preserved).

Acceptability Relation. The acceptability relation is a formal way to abstract away the correspondence between program states in different languages: Recall that the cut-bisimulation theory is parameterized by a given acceptability relation, which relates equivalent states across the two programs. Such correspondence may not be trivial between two different languages. However it has no effect on the theory other than the requirement that the states related in the cut-bisimulation relation must also be related in the acceptability relation. In general, the acceptability relation can be arbitrarily complex to define for any two given different languages. In our system, we provide `common.k`, a third semantic definition accepted by Keq, for formally defining complex acceptability relations. This way, we can at least make the formal definitions for the same language pairs reusable, similar to the language semantic definitions themselves. In the case of LLVM IR and Virtual x86, the acceptability relation is mostly straight-forward. Specifically, in our TV system, the `common.k` module contains various definitions of equivalent (or common) components of the state in the two languages. This serves as a shortcut so that these components need not be repeatedly marked as equivalent in every synchronization point. The most significant such component is the memory model used for the two definitions (see Subsection 4.4).

Keq runs the equivalence checking algorithm presented in Section 3 on the given set of synchronization points and outputs a verdict that validates the translation instance or flags it as not validated. In our ISel TV system, we need to trust (beyond the Keq implementation and the $\mathbb{K}$ semantic definitions) that the given synchronization points cover all entry points of each function of the program, and the synchronization points belong to the acceptability relation.[6] Note that we do *not* need to trust that other relevant points (e.g., exit points, loopheads, etc.) are also covered by the set of synchronization points, because otherwise Keq will fail.

4.1 LLVM Instruction Selection Phase

The ISel phase [23] of an LLVM-based compiler is responsible for translating LLVM IR into a selected target's instruction set. This is a non-trivial component of the compiler, implemented in more than 140,000 lines of C++ and TableGen [25] code (*excluding* target-specific code for back-end targets other than x86-64). During ISel, LLVM IR code is first converted into a target-independent directed acyclic graph (DAG) representation called SelectionDAG with one DAG per basic block. Next, the instruction selection happens by matching patterns of DAG subgraphs to new subgraphs that contain the target opcodes. Finally, the DAG nodes are linearized to produce instruction sequences per basic block.

[6] In principle, the latter can be excluded from the trust base by verifying it in a separate process, but we only manually checked it since the acceptability relation is rather straightforward in this case.

Our translation validation prototype works for LLVM version 5.0.2. There are two different instruction selection algorithms in LLVM 5.0.2: SDISel and FastISel. SDISel is slower and more sophisticated, and is the default for compilation to native; FastISel is faster and used at O0 and in JIT compilation. Our prototype works on SDISel with -O0, since this level performs the least amount of extra transformations to the code, focusing instead mainly on the language translation. Optimizations enabled in higher levels include more aggressive pattern folding and more aggressive constant propagation. These do not affect the applicability of our method, but may require more sophisticated hint generation.

4.2 LLVM IR Semantics

The LLVM IR documentation can be found in [22]. In our LLVM semantics definition, we model the `i1`, `i8`, `i16`, `i32`, and `i64` integer types, composite (arbitrarily nested) array and struct types, the corresponding pointer types, and type-cast instructions, including integer/pointer casts (`inttoptr` and `ptrtoint`). The `getelementptr` instruction is used to compute the address of an element within a composite type. We also model integer arithmetic operators, bitwise operators, and the integer and pointer comparison operators. We model the control flow instructions for unconditional and conditional branches, as well as function calls and returns. Finally, the supported memory operations are loads, stores, and the `alloca` instruction for stack allocation of local variables. The LLVM semantics uses the common memory model described in Subsection 4.4 as its memory abstraction. Our memory abstraction does not yet take alignment requirements into consideration, so we do not support programs that assume any kind of variable or load/store alignment.

4.3 Virtual x86 Semantics

The output of ISel is LLVM Machine IR, a low-level representation representing the opcodes and operand types of the selected target ISA. More specifically, the LLVM Machine IR is a register-based IR that is parametric to any number of ISA opcodes and physical registers. It also supports a number of higher-level features such as various pseudo-opcodes (such as `COPY`, `PHI`, and others), an unlimited number of virtual registers, a frame abstraction for modeling call stacks, and a jump table abstraction. The Machine IR used in the x86 backend of the LLVM compiler is then specialized by using all the x86 opcodes and the full x86 physical register file [15]. We call this version of the Machine IR "Virtual x86."

Our $\mathbb{K}$ semantic definition of Virtual x86 captures all the extended features except jump tables and also various features of x86-64. We model integer arithmetic operations and integer comparison operations, the general-purpose physical registers, conditional and unconditional jump instructions, and the flags and program counter registers, `eflags` and `rip`. As with the LLVM IR semantics, the program address space is modeled using the common memory model abstraction described in Subsection 4.4. We model various move instructions that copy data between registers and memory.

4.4 Common Memory Model

Both the LLVM and Virtual x86 semantics definitions use a common, low-level, sequentially consistent memory model. This simplifies the formal definition of equivalent memory configurations between LLVM and Virtual x86 programs. The semantic definition of this common memory model is part of the acceptability relation and is contained in `common.k` (see Section 2 and Figure 5).

4.5 Characterizing Synchronization Points

Each synchronization point is a pair of symbolic states of the input and output programs, accompanied by a set of equality constraints over symbolic variables in the two states. Figure 3 shows the synchronization points generated by our TV system for the programs in Figure 2. For example, the synchronization point p1 consists of a symbolic state for the LLVM IR program that represents states entering the `for.cond` basic block from the `entry` basic block, and a symbolic state for the Virtual x86 program that represents states entering the `.LBB1` basic block from the `.LBB0` basic block. In addition, the point represents only pairs of states that satisfy the equality constrains `%d = %vr7_32`, `%a.0 = %vr6_32`, `%n = %vr8_32`, and `%vr9_32 = 1`, where the names of the virtual registers serve also as the names of the symbolic values that they hold in the corresponding symbolic states. In general, each synchronization point describes a potentially infinite number of input and output program state pairs, one pair for each concrete substitution of the symbolic variables of the two states that satisfies both state constraints as well as the equality constraints of the synchronization point.

To be sound, the synchronization points should be a cut for the programs, i.e., "covering" all possible program executions (see Section 2). In the rest of this subsection, we discuss how the synchronization point generator creates the set of points to be given to Keq using compiler-provided hints and static analysis results. Please note that this generator is specifically designed for the ISel pass of LLVM. As discussed earlier, different transformations may require different synchronization point generation strategies.

Function Granularity. An important design decision is whether input/output function pairs should be treated independently or not. When function pairs are treated independently, the translation of each function is considered a unique instance of the equivalence checking problem. In this case, we can assume that both caller and callee functions will be translated correctly, and hence function calls to the same function are equivalent. This is the same assumption that the compiler makes when applying any intraprocedural transformation to a function, and for this reason treating function pairs independently is a natural choice when doing translation validation of an intra-procedural transformation, such as ISel. This approach has the added benefit that it deals uniformly with cases of missing code and compile-time unknown callers and callees: Every function call, whether the callee is known, unknown, or missing is treated the same. Only the missing callers and callees must be trusted because translations of all available functions will be validated.

Function Entry and Exit. We generate synchronization points that cover the entry and corresponding exits of each function pair. These are the points $p0$ and $p3$ in Figure 3 for the program in Figure 2. We can infer the equality constraint for these points from the calling convention.

Loop Entry. We also generate synchronization points that cover the entries of corresponding loops in order to cover states that belong to cycles. These are the points $p1$ and $p2$ in Figure 3 (one point per predecessor to expedite the symbolic execution of the phi instructions). The relation between loops in the input and output is provided as a compiler-generated hint. The equality constraints for these points relate corresponding live registers in the input and output. The register correspondence is provided as a compiler-generated hint, computing using a Live Variables static analysis.

Call sites. Assuming that calls starting from equivalent states result in returns to equivalent states, it suffices to generate synchronization points before and after corresponding call sites. A synchronization point before a call site is treated as covering an exiting state (meaning that we do not symbolically execute the call itself in Keq). The constraints for a point after a call site relate corresponding live registers and corresponding return values. The equality constraints for the points before and after a call site are inferred from the calling convention.

Memory state. Finally, all synchronization points should contain constraints that ensure that corresponding memory objects accessible by the functions hold the same contents. Since our prototype uses a common memory model for input and output, this requirement is translated to a simple equality constraint between the whole memory of the input and output.

In summary, the only compiler generated hints required in our approach are pairs of corresponding LLVM and Virtual x86 virtual registers and pairs of corresponding loops. The hint generator records and outputs these for each translation instance. Its implementation is trivial, adding just about 500 lines of C++ code to ISel, and does not require any formal methods expertise.

4.6 Characterizing Undefined Behaviors

The ability to handle undefined behaviors is an important aspect of a practical TV system for C/C++ programs. Our prototype handles undefined behaviors related to memory out-of-bounds accesses as well as signed integer overflow.

We model these undefined behaviors by a set of uniquely marked error states. When such behavior can potentially happen in a symbolic state, our semantics have rules to conditionally branch into an error state, which captures information about the nature of the reached undefined behavior. Our acceptability relation for LLVM IR and Virtual x86 relates LLVM error states to any Virtual x86 state. On the other hand, Virtual x86 error states are only related with relevant LLVM error states in the acceptability relation, e.g. the out-of-bounds access error state in Virtual x86 is related only to the out-of-bounds access error state in LLVM. This way Keq automatically reverts to checking refinement in the presence of undefined behaviors.

4.7 Challenging Validations

Although, the TV system presented here is able to validate a large number of real-world code compilations (see Section 5), we briefly discuss a few examples that have proved challenging:

- Modeling undefined behaviors in LLVM (in particular `poison` and `undef` values) requires much more work; our method only works for simple undefined behaviors that can be captured by transitioning to a single "error" state. Lee *et al.* [20] gives a detailed analysis of undefined behavior in LLVM and the challenges of formally modeling it.
- ISel performs some strength reductions that are difficult to prove correct in Z3. For example, LLVM sometimes reduces a division by a constant to shifts and multiplications, which would take Z3 a considerable amount of time to prove without adding dedicated lemmas.

Result	#Functions
Succeeded	4,331
Failed due to timeout	206
Failed due to out-of-memory	179
Other	16
Total	4,732

Table 1: Translation validation results for GCC benchmark

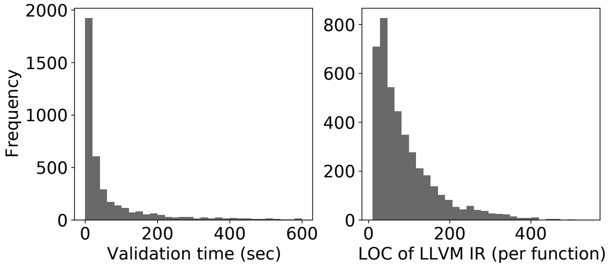


Figure 6: Distributions of validation time and code size

5 EVALUATION ON REAL-WORLD CODE

We evaluate the Translation Validation system for Instruction Selection in two ways. First, we apply the system to the compilation of the GCC SPEC 2006 benchmark [37]. Second, we reintroduce several bugs that were found and fixed in the Instruction Selection pass of LLVM and verify that our system does not validate translations that trigger these bugs, which are miscompilations.

5.1 Application to GCC from SPEC 2006

We applied the Translation Validation prototype to the source code of the GCC SPEC 2006 benchmark, a version of an important piece of software that affects the correctness of many other critical software systems (e.g., the Linux kernel). The GCC source code is comprised of 5572 C functions, which we compiled to LLVM IR using `clang-5.0.2` at optimization level -O0 and translated to Virtual x86 by the ISel pass of LLVM 5.0.2. For each verification run, we allocated 2 Intel Xeon CPU E7-8837 processors at 2.67GHz and 12GB of memory, with a timeout of 3 hours.

Out of the 5572 functions, our evaluation considered 4732 functions that are covered by our LLVM and x86 language semantics (Section 4.2 & 4.3). The remaining functions involve floating point, SIMD, or certain bitwise operations that are not supported by the current semantics. In the following discussion, 4732 will be the denominator of all percentages mentioned.

Figure 6 shows distributions of validation time and the code size of the functions. With the above hardware setup, the average time for processing a function in our GCC experiment is 150 seconds and the median is 0.8 seconds. Note that this does not include the time used for K to load the semantics and parse the input proofs.

Out of the 4732 functions, our prototype was able to formally verify the translation of 4331 functions (91.52%). Table 1 categorizes the reasons for failure for the remaining 401 functions. We discuss these categories in more detail below.

Timeout. 206 functions (4.35%) failed due to timeout (3 hour limit), and the Z3 solving time was the dominating factor. With the symbolic variant of Algorithm 1, Keq may make multiple Z3 queries in each step containing path conditions, which grow significantly over time, particularly when there is a large number of complicated memory operations and branching conditions. Making things worse, the current integration of Z3 in the 𝕂 framework does not use the incremental query solving feature of Z3, and thus solving each query needs to have a cold start even if many of the complex queries share significant sub-queries. We believe that this can be improved by integrating the incremental Z3 query solving to the 𝕂 framework.

Out of memory. 179 functions (3.78%) failed with an out-of-memory exception. All these failures happened during the parsing of our synchronization point specifications, which were caused by performance issues in the 𝕂 builtin parser. The 𝕂 parser was designed to be quite general, equipping a comprehensive parsing ambiguity resolving mechanism, which does not often scale well. This can be improved by using a more compact representation of the synchronization point specification to alleviate the burden on the 𝕂 parser, or by using a more recent feature of 𝕂 to generate a static parser ahead of time.

Inadequate synchronization points. The rest of the failures (16 functions) are due to an inaccuracy in our liveness analysis, that resulted in a mismatch of LLVM and Virtual x86 live registers at the beginning of certain basic blocks, i.e. a live register in the x86 block with no live counter-part in the LLVM block. This caused the VC generator to generate an inadequate set of synchronization points for an equivalence proof. A more sophisticated liveness analysis would resolve this issue.

5.2 Evaluation with Real LLVM Bugs

We discuss two Instruction Selection bugs previously reported in the LLVM code base. Although the bugs are currently fixed, we were able to reintroduce them in the compiler and we attempt to validate translations that trigger the buggy code with our system. In both cases, the Translation Validation system failed to verify the buggy translation, which is the desired outcome.

Write-after-write dependency violation when translating store instructions. This bug causes a miscompilation that violates a write-after-write dependency for a memory location when subsequent overlapping stores access said location. The compiler erroneously reorders the two write accesses while attempting to optimize the compilation of the store instructions by merging them into fewer wider stores.

This is a bug for the x86-64 backend and it last appeared in clang 3.7.x (as a regression from older versions) for optimization levels -O2 and -O3 [2]. Figures 7 and 8 demonstrate the miscompilation. Figure 7 shows the LLVM code. The shown function performs 3 2-byte wide stores at offsets 2, 3, 1 of a global byte array. This means that the first two stores both write the byte at offset 3. A straightforward correct translation to x86-64 is shown in Figure 8(a), while Figure 8(c) shows a correct optimized compilation: The third store has been merged into the first store that becomes a 4-byte wide store. This is correct because there is no dependency between the third store and any of the rest and the order of the first and second store has been preserved. On the other hand, Figure 8(b) shows the miscompilation due to the bug: This time the first store has been merged into the third thus reversing the write-after-write dependency between the first and second store.

Our system catches the bug, since Keq cannot prove the candidate synchronization point set is a cut-bisimulation. Indeed, starting from the entry point and assuming that the global memory contents are the same, the symbolic execution of the input and output programs leads to different memory contents for the byte at offset 3, hence not allowing Keq to prove the constraint for equal memory contents at the exiting synchronization point.

Incorrect load narrowing with non-power-of-two types. This bug causes a miscompilation that leads to an out-of-bounds memory access. The compiler erroneously compiles a 4-byte wide load to an 8-byte wide load when attempting to narrow a load that accesses a memory location holding a non-power-of-two bitwidth type.

This is a bug for the x86-64 backend and it was found in clang 2.6.x for optimization levels -O2 and higher [1]. Figure 9 and 10 demonstrate the miscompilation. Figure 9 shows the LLVM code. The shown function loads from a memory location holding a 12-byte (`i96`) integer. It then logically shifts right the lower 8 bytes and stores the remaining 4 bytes, zero-extended as an 8-byte (`i64`) integer to another memory location. A correct translation to x86-64 is shown in Figure 10(a): The code first loads the upper 4 bytes of the source location into `eax`, thus zeroing-out the higher 4 bytes of the 64-bit general purpose register `rax`, according to the x86-64 semantics. It then stores `rax` which now holds the correct contents (the higher 4 bytes of the store zero-extended as an 8-byte integer) to the destination memory location. On the other hand, Figure 10(b) shows the miscompilation due to the bug: This time the load is 8-byte wide, thus accessing 4 out-of-bounds bytes that may cause a segmentation fault. Furthermore, the value stored at the destination may be incorrect because the 4 higher bytes are not zeroed-out but rather have random values.

Similar to previous case, our system catches the bug, since Keq cannot prove the candidate synchronization point set is a cut-bisimulation. Indeed, starting from the entry point and assuming that the global memory contents are the same, the symbolic execution of the output x86 program branches into an out-of-bounds error state in addition to reaching the exiting point. This error state cannot be matched with any state in the input LLVM program, hence not allowing Keq to prove cut-bisimulation: there is a behavior in the output that is not found in the input[7].

[7] In fact, in this case we cannot even prove that the output refines the input.

```
@b = external global [8 x i8]

define void @foo() {
entry:
  store i16 0, i16* bitcast (i8* getelementptr inbounds ([8 x i8], [8 x i8]* @b, i64 0, i64 2) to i16*)
  store i16 2, i16* bitcast (i8* getelementptr inbounds ([8 x i8], [8 x i8]* @b, i64 0, i64 3) to i16*)
  store i16 1, i16* bitcast (i8* getelementptr inbounds ([8 x i8], [8 x i8]* @b, i64 0, i64 0) to i16*)
  ret void
}
```

Figure 7: LLVM IR - write-after-write dependency violation

```
foo:
  movw $0, b+2(%rip)
  movw $2, b+3(%rip)
  movw $1, b(%rip)
  retq
```

(a) *Simple correct translation*

```
foo:
  movw $2, b+3(%rip)
  movl $1, b(%rip)
  retq
```

(b) *Optimized incorrect translation*

```
foo:
  movl $1, b(%rip)
  movw $2, b+3(%rip)
  retq
```

(c) *Optimized correct translation*

Figure 8: x86 - write-after-write dependency violation

```
@a = external global i96, align 4
@b = external global i64, align 8

define void @foo() {
  %srcval = load i96, i96* @a, align 4
  %tmp96 = lshr i96 %srcval, 64
  %tmp64 = trunc i96 %tmp96 to i64
  store i64 %tmp64, i64* @b, align 8
  ret void
}
```

Figure 9: LLVM - load narrowing with non-power of 2 types

```
foo:
  movl a+8(%rip), %eax
  movq %rax, b(%rip)
  retq
```

(a) *Optimized correct translation*

```
foo:
  movq a+8(%rip), %rax
  movq %rax, b(%rip)
  retq
```

(b) *Optimized incorrect translation*

Figure 10: x86 - load narrowing with non-power of 2 types

6 RELATED WORK

Verified Compilers: One approach to the problem of compiler verification is the full formal verification of the compiler, as in CompCert [21], CakeML [17], and the lambda calculus to typed assembly compiler in [4]. Also formal verification of specific compiler transformation passes, e.g., SSA-based transformations [43] and peephole optimizations [26], has been proposed. Full formal verification is attractive because it gives an ahead-of-time guarantee of correctness for all input programs, whereas TV approaches detect errors only when actually compiling programs and are also susceptible to false alarms. Ahead-of-time verification cannot encounter such an error, but may suffer if certain optimizations cannot be proven correct, leading to weaker optimization choices and consequently possibly poorer performance of the generated code. So far, this approach has only been used for compilers built from the ground up with the goal of verification in mind. For example, CompCert [21], a verified compiler for C, has been written in the Coq Proof Assistant's specification language. The approach requires extensive manual effort ("proof engineering"), and much greater expertise in formal methods than is usually available in production compiler teams. Such design decisions and development processes appear impractical to apply retroactively in existing compilers not specifically designed for full verification.

TV Systems: Translation Validation as a method of verifying the correctness of a compilation was first proposed by Samet [33] and reformulated by Pnueli *et al.* [30]. TV has been used to prove correctness of specific compiler optimization passes [18, 28, 29, 36, 40, 41, 44], discover compiler bugs [13], and to prove correctness of end-to-end compilation [30, 35]. VOC-64 [44] for the SGI Pro-64 compiler, Necula *et al.* [29] for the GNU C compiler, Peggy [40] for the Soot Java bytecode optimizer, LLVM-MD [41] and Namjoshi *et al.* [28] for LLVM IR passes, are all tools that perform translation validation for the respective optimization passes in production compilers. Sewell *et al.* [35] presents a TV approach for the compilation of the seL4 kernel from C to binary. Hawblitzel *et al.* [13] use a TV approach to determine whether assembly code produced by different versions of the CLR JIT compiler are semantically equivalent and thus report miscompilations when there are differences. PEC [18] proves the correctness of optimizations by applying TV techniques in pairs of partially specified programs, where such a pair describes a general optimization on all the corresponding concrete programs. DDEC [36] is an equivalence checker for x86 loops that uses data collected from test runs rather than inference or hints to construct a simulation relation.

The proof of program equivalence in the majority of these TV tools [13, 18, 29, 30, 35, 36, 44] is based on generating sets of verification conditions, the satisfiability of which is enough to prove equivalence. The VCs are produced as a combination of invariants that have to be inferred and a refinement requirement that is defined in a slightly different way in the context of each work. All these various refinement requirements attempt to capture a certain weak simulation variant. We claim that cut-bisimulation, another weak bisimulation variant, is more suitable for compiler translation validation in practice, and that in fact, any of these refinement requirements can be expressed as a cut-simulation proof requirement. For instance, the equivalence proof rule used to generate the refinement requirement in VOC-64 [44] is reminiscent of our notion of cut-similarity, but is expressed using syntactic devices (such as basic blocks and paths in the control flow graph) that unnecessarily restrict its generality and distance it from classic bisimulation theory. Again, we do *not* claim that the motivating idea of cut-bisimulation is new, but only that our formulation captures the essential properties required for compiler translation validation, and moreover, enables proof systems to be parameterized by the operational semantics of the input and output languages.

Namjoshi *et al.* [28] uses a variant of stuttering-bisimulation with ranking functions, first introduced in [27], which informally represent how many times one of the transition systems is allowed to stutter while the other advances before the former has to advance. This variant requires matching single transitions only, similarly to strong bisimulation and unlike classic stuttering bisimulation, where a single transition may have to be matched with a finite but unbounded number of transitions, thus leading to large number of generated proof requirements. Cut-bisimulation shares the same property of matching single transitions only and is more appealing for proof automation, since the proof generator does not need to produce ranking functions in addition to synchronization points.

Finally, LLVM-MD [41], Peggy [40] and Dasgupta *et al.* [8] move away from simulation proofs, and instead use graph isomorphism techniques to prove equivalence.

All the previous work on Translation Validation we know of assumes that the input and output programs are either written in or are translated to a common language or representation: GNU RTL [29], LLVM IR [28], value graphs [40, 41], x86 [36], Boogie IR [3, 13], a C-like intermendiate language for PEC [18], and a common representation called Transition Systems [30, 44]. Even the translation validation approach for the seL4 kernel proposed in [35] requires translation of the input C code and decompilation of the output binary to a common graph language used for equivalence checking. This requires trusting unchecked translations into a common language, which is not much simpler in complexity than the original translation itself. On the other hand, our equivalence checking algorithm is parametric to the input and output language semantics, thus generalizing the original approach of Pnueli *et al.* by eliminating the requirement for a common semantic framework. This makes it much easier (and more robust) to validate translations between two different languages (e.g., as in Instruction Selection), because it does not require carrying out (and trusting) the unchecked translations. Keq is the first tool for program equivalence checking we know of that is truly language-independent.

Hints vs Heuristics for VC generation: Our proposed algorithm takes as input a relation between program points in the input and output languages. To generate this relation, our implemented prototype for LLVM ISel uses compiler-generated hints, similar to the witnesses introduced in [28]. Other works discuss various heuristics that can be used instead. In particular Necula *et al.* [29] describes an inference algorithm to generate both, a relation between program points and the accompanying constraints between program variables and memory locations for two functions when any number of compiler transformations have been applied to the original function to produce the transformed function. The algorithm uses transfer functions to describe the effect of each basic block, which are generated automatically using symbolic execution. Working towards a language independent proof generator, it is possible that one can derive a language independent version of this inference algorithm by implementing to be parametric in the language semantics in a fashion similar to Keq. Finally, DDEC [36] uses a combination of static analysis and data-driven inference for constructing simulation relations: Static analysis is used to determine the program locations of synchronization points and the live variables while the constraints between variables are inferred from execution traces.

Mutual Equivalence Proof System: Our equivalence checking algorithm was inspired from the language-independent proof system for mutual equivalence introduced in [5]. Instead of a proof system, here we propose a bisimulation relation and an algorithm based on it and symbolic execution, leading to the first language-independent implementation of a checker for equivalence between programs written in two different languages.

7 FORMALIZATION OF CUT-BISIMULATION

Notations. Given a binary relation $R \subseteq S_1 \times S_2$, we write $a\,R\,b$ to denote $(a, b) \in R$; and $R_1 = \{a \mid \exists b\,.\,a\,R\,b\}$ and $R_2 = \{b \mid \exists a\,.\,a\,R\,b\}$ to denote the projections $\Pi_i(R)$ for $i \in \{1, 2\}$.

Let $\mathcal{S}$ be a set of states (thought of as all possible states of a language, over all programs in the language). Let $T = (S, \xi, \rightarrow)$ be an $\mathcal{S}$-*transition system*, or just a *transition system* when $\mathcal{S}$ is understood, that is a triple consisting of: a set of *states* $S \subseteq \mathcal{S}$, an *initial state* $\xi \in S$, and a (possibly nondeterministic) *transition relation* $\rightarrow \subseteq S \times S$. Let $next(s)$ denote the set $\{s' \mid s \rightarrow s'\}$. T is *finitely branching* iff $next(s)$ is finite for each $s \in S$. Let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$, and $\rightarrow^+$ be the transitive closure of $\rightarrow$.

A (possibly infinite) *trace* $\tau = s_0 s_1 \cdots s_n \cdots$ is a sequence of states with $s_i \rightarrow s_{i+1}$ for all $i \geq 0$. Let $\tau[n]$ be the n^{th} state of τ where the index starts from 0, and let $\mathsf{size}(\tau)$ be the length of τ (∞ when τ is infinite). Let $\mathsf{first}(\tau) = \tau[0]$ be the first state of τ, and let $\mathsf{final}(\tau)$ be the final state of τ when τ is finite. Let $\mathsf{traces}(s)$ be the set of all traces starting with s, also called s-*traces*, and let $\mathsf{traces}(S)$

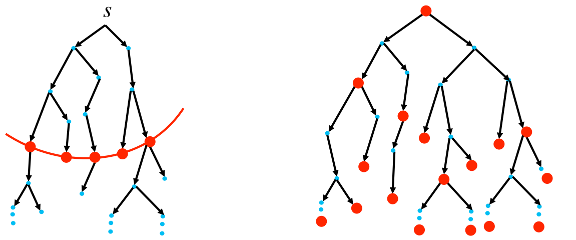


Figure 11: Left: a cut C for state s (each complete s-trace intersects C). Right: a cut C for a transition system (C contains the initial state and is a cut for itself, i.e., for each state in C)

be $\bigcup_{s \in S}$ traces(s). A *complete trace* is either an infinite trace, or a finite trace τ where *next*(final$(\tau)) = \emptyset$.

Definition 7.1 (Cut and Cut Transition System). *Let $T = (S, \xi, \longrightarrow)$ be a transition system. A set $C \subseteq S$ is a* cut for $s \in S$, *iff for any complete trace $\tau \in$ traces(s), there exists some strictly positive $k > 0$ such that $\tau[k] \in C$. The set $C \subseteq S$ is a* cut for T *iff $\xi \in C$ and C is a cut for each $s \in C$, in that case T is called a* cut transition system *and is written as a quadruple $(S, \xi, \longrightarrow, C)$. See Figure 11.*

In a cut transition system, any finite complete trace starting with the initial state terminates in a cut state, and any infinite trace starting with the initial state goes through cut states infinitely often:

Lemma 7.2. *Let $T = (S, \xi, \longrightarrow, C)$ be a cut transition system. Then for each complete trace $\tau \in$ traces(ξ) and each $0 < i <$ size(τ), there is some $j \geq i$ such that $\tau[j] \in C$.*

Proof. Let $\tau \in$ traces(ξ) be a complete trace. Assume to the contrary that there exists i such that $\forall j \geq i.\ \tau[j] \notin C$. Pick such an i. Then we have two cases. When $\forall k < i.\ \tau[k] \notin C$, we have $\forall k > 0.\ \tau[k] \notin C$, which is a contradiction since C is a cut for $\xi = \tau[0]$. Otherwise, $\exists k < i.\ \tau[k] \in C$, and let k be the largest such number. Then, we have $\forall l > k.\ \tau[l] \notin C$, which is a contradiction since C is a cut for each $s \in C$, thus a cut for $\tau[k] \in C$. □

Cuts do not need to be minimal in practice, and are not difficult to produce. For example, a typical cut includes all the final states (normally terminating states, error/exception states, etc.) and all the states corresponding to entry points of cyclic constructs such as loops and recursive functions. Such cut states can be easily identified statically using control-flow analysis, or dynamically using an operational semantics.

Definition 7.3 (Cut-Successor). *Let $T = (S, \xi, \longrightarrow, C)$ be a cut transition system. A state s' is an (immediate)* cut-successor *of s, written $s \rightsquigarrow s'$, iff there exists a finite trace $ss_1 \cdots s_n s'$ where $s' \in C$ and $n \geq 0$ and $s_i \notin C$ for all $1 \leq i \leq n$.*

Definition 7.4 (Cut-Bisimilarity). *Let $T_i = (S_i, \xi_i, \longrightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). Relation $R \subseteq C_1 \times C_2$ is a* cut-simulation *iff whenever $(s_1, s_2) \in R$, for all s_1' with $s_1 \rightsquigarrow_1 s_1'$ there is some s_2' such that $s_2 \rightsquigarrow_2 s_2'$ and $(s_1', s_2') \in R$. Let $\leq$ be the union of all cut-simulations (also a cut-simulation). Relation R is a* cut-bisimulation *iff both R and R^{-1} are cut-simulations. Let $\sim$ be the union of all cut-bisimulations (also a cut-bisimulation).*

Cut-bisimulation generalizes classic (strong) bisimulation [34]. A cut-bisimulation on $(S_i, \xi_i, \longrightarrow_i, C_i)$ is a bisimulation on $(S_i, \xi_i, \longrightarrow_i)$, when $C_i = S_i$. Furthermore, a cut-bisimulation on T becomes a bisimulation on the cut-abstract transition system of T, as described below.

Definition 7.5 (Cut-Abstract Transition System). *Let T be a cut transition system $(S, \xi, \longrightarrow, C)$. The* cut-abstract transition system *of T, written $\overline{T}$, is the (standard) transition system $(C, \xi, \rightsquigarrow)$.*

Lemma 7.6. *Let $T_i = (S_i, \xi_i, \longrightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). A relation $R \subseteq C_1 \times C_2$ is a cut-bisimulation on T_1 and T_2, iff R is a (standard) bisimulation on $\overline{T_1}$ and $\overline{T_2}$.*

Proof. By identifying $\rightsquigarrow_i$ as the transition relation of $\overline{T_i}$. □

Corollary 7.7. *Let $T_i = (S_i, \xi_i, \longrightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). Let R be a cut-bisimulation, and $(s_1, s_2) \in R$. For any state $s_1' \in C_1$ with $s_1 \longrightarrow_1^+ s_1'$, there exists some $s_2' \in C_2$ with $s_2 \longrightarrow_2^+ s_2'$ such that $(s_1', s_2') \in R$. The converse also holds.*

Proof. By Lemma 7.6 and the bisimulation invariant of reachability. □

Now we formalize the equivalence of cut transition systems in the presence of a given acceptability (or compatibility, or indistinguishability) relation $\mathcal{A}$ on states.

Definition 7.8. *Let $\mathcal{A} \subseteq S_1 \times S_2$, which we call an* acceptability relation. *Let $T_i = (S_i, \xi_i, \longrightarrow_i, C_i)$ be two cut transition systems ($i \in \{1, 2\}$). T_2* cut-simulates *T_1* (i.e., *T_1* cut-refines *T_2*) w.r.t. *$\mathcal{A}$, written $T_1 \leq_{\mathcal{A}} T_2$, iff there exists a cut-simulation $\mathsf{P} \subseteq \mathcal{A}$ such that $\xi_1\ \mathsf{P}\ \xi_2$. Furthermore, T_1* and *T_2* are cut-bisimilar w.r.t. *$\mathcal{A}$, written $T_1 \sim_{\mathcal{A}} T_2$, iff there exists a cut-bisimulation $\mathsf{P} \subseteq \mathcal{A}$ such that $\xi_1\ \mathsf{P}\ \xi_2$.*

Note that if a cut-bisimulation P like above exists, then there also exists a largest one; that's because the union of cut-bisimulations included in $\mathcal{A}$ is also a cut-bisimulation included in $\mathcal{A}$. We let the relation $\sim_{\mathcal{A}}$ denote that largest cut-bisimulation, assuming that it exists whenever we use the notation (and similarly for $\leq_{\mathcal{A}}$).

Our thesis is that $\sim_{\mathcal{A}}$ yields the right notion of program equivalence. That is, that two programs are equivalent according to a given state acceptability (or compatibility or indistinguishability) relation $\mathcal{A}$ between the states of the respective programming languages, iff for any input, the cut transition systems T_1 and T_2 corresponding to the two program executions satisfy $T_1 \sim_{\mathcal{A}} T_2$. The following result strengthens our thesis, stating that cut-bisimilar transition systems reach compatible cut states, and, furthermore, that they cannot indefinitely avoid reaching a cut state:

Theorem 7.9. *If $T_1 \sim_{\mathcal{A}} T_2$ then for each s_1 with $\xi_1 \longrightarrow_1^+ s_1$ there exists some s_2 with $\xi_2 \longrightarrow_2^+ s_2$, such that: (1) if $s_1 \in C_1$ then $s_1 \sim_{\mathcal{A}} s_2$; and (2) if $s_1 \notin C_1$ then there exists some $s_1' \in C_1$ such that $s_1 \longrightarrow^+ s_1'$ and $s_1' \sim_{\mathcal{A}} s_2$. The converse also holds.*

Proof. We only need to show the forward direction, since the backward is dual. First we have $\xi_1 \sim \xi_2$ by Definition 7.8 and the fact that $\sim$ is the union of all cut-bisimulations. Let s_1 be a state with $\xi_1 \longrightarrow_1^+ s_1$. Then we have two cases:

- When $s_1 \in C_1$. There exists s_2 such that $\xi_2 \longrightarrow_2^+ s_2$ and $s_1 \sim s_2$ by Corollary 7.7.

- When $s_1 \notin C_1$. There exists s_1' such that $s_1 \to_1^+ s_1'$ and $s_1' \in C_1$ by Lemma 7.2 and the fact that C_1 is a cut for $\xi_1 \in C_1$. Then, there exists s_2 such that $\xi_2 \to_2^+ s_2$ and $s_1' \sim s_2$ by Corollary 7.7.

□

8 CORRECTNESS OF EQUIVALENCE CHECKING ALGORITHM

We show that Algorithm 1 is refutation-complete, i.e. if it does not terminate then the two programs are equivalent:

Theorem 8.1 (Correctness of Algorithm 1). *Suppose that cut transition systems $T_i = (S_i, \xi_i, \to_i, C_i)$ are finitely branching ($i \in \{1, 2\}$) and $\mathsf{P} \subseteq \mathcal{A}$ is recursively enumerable with $(\xi_1, \xi_2) \in \mathsf{P}$. If Algorithm 1 does not terminate with false, then $T_1 \sim_{\mathcal{A}} T_2$.*

Proof. By Definition 7.8, we only need to show that P is a cut-bisimulation when `main` does not terminate with returning false. First let us characterize the two sub-functions:

(1) $\texttt{check}(p_1, p_2)$ terminates; and if it returns true, then for all $p_1 \rightsquigarrow_1 s_1$, there exists s_2 such that $p_2 \rightsquigarrow_2 s_2$ and $(s_1, s_2) \in \mathsf{P}$; and the converse also holds.

(2) $\texttt{next}_i(n)$ terminates and returns the set of all cut-successors of n, i.e., $\{n' \mid n \rightsquigarrow_i n'\}$.

P is cut-bisimulation by (1), while the claim (1) is straightforward by (2). Now we only need to show the claim (2). To show (2), let us claim the invariant of the while loop as follows. It is easy to show that it is maintained in each iteration.

- For each finite trace $n_1 \cdots n_k n'$ ($k \geq 1$) such that $n_1 = n$, $n_j \notin C_i$ ($1 < j \leq k$) and $n' \in C_i$, either holds: $n' \in Ret$ or $\exists m \in [1, k].\ n_m \in N$.
- For each $n' \in N \cup Ret$ such that $n' \neq n$, there exists a finite trace $n n_1 \cdots n_k n'$ ($k \geq 0$) such that $n_j \notin C_i$ ($1 \leq j \leq k$).
- $Ret \subseteq C_i$.

Now let us show (2). First, $Ret \subseteq \{n' \mid n \rightsquigarrow_i n'\}$ by the second and the third bullets. Then, $Ret \supseteq \{n' \mid n \rightsquigarrow_i n'\}$ by the first bullet, since N is empty when it terminates. Let us show the termination. Assume $\texttt{next}_i(n)$ does not terminate. If it does not terminate, then since T_i is finitely branching, there should exist an infinite trace from n that never comes across one of C_i, which is the contradiction since C_i is a cut for $n \in \mathsf{P}_i \subseteq C_i$. Now let us show that the loop invariant is maintained in each iteration. The second and the third bullets are trivial. Let us show the first bullet. Assume that it holds in some iteration. Pick such a finite trace $n_1 \cdots n_k n'$. We have three cases: $n' \in Ret$, $n_k \in N$, and $n_m \in N$ where $m < k$. For the first and the third cases, it is easy to show the invariant is maintained in the next iteration. In the second case, we have $n' \in \text{next}(n_k)$ by definition of the traces. Then n' is added to Ret in line 23, since $n' \in C_i$, and we conclude. □

Note that if we replace the if-condition of line 11 with "$\forall n \in N_1.\ n.\text{color} = \text{black}$", then it suffices to show $T_1 \leq_{\mathcal{A}} T_2$, i.e., T_1 refines T_2.

9 CONCLUSION

We have presented a novel algorithmic approach for proving cross-language program equivalence. Our algorithm relies on *cut-bisimulation*, a general formalization of weak bisimulation relations that is better suited to equivalence proofs when the two programs may have unrelatable intermediate states. Two out of three key components of our equivalence checking algorithm – the formal notion of program equivalence, and the proof system that generates and checks equivalence proofs for given verification conditions – are entirely independent of specific transformations. This enables the reuse of a significant part of the TV system and allows developers to focus on the important aspects of the problem, which are the semantic definitions of the input and output languages and the generation of the proof obligations. Moreover, cut-bisimulation allows for generation of proof obligations that take the intuitive (from a compiler's perspective) form of synchronization points between the input and output programs. We have used this algorithm to develop the first *language-independent* tool for program equivalence, and a prototype TV system for a major translation phase of the LLVM compiler infrastructure, instruction selection. In the future, we aim to leverage the work presented in this paper to develop an end-to-end TV system for LLVM-based compilers.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1619275, and by the Office of Naval Research under Grant No. N00014-17-1-2996. We thank our shepherd, Sara Achour, for her diligent review of preliminary versions of the paper. Sandeep Dasgupta, Sean Bartell, John Regehr, and the anonymous reviewers provided valuable feedback.

REFERENCES

[1] 2015. Instruction Selection load narrowing miscompilation. https://bugs.llvm.org/show_bug.cgi?id=4737.
[2] 2015. Instruction Selection WAW miscompilation. https://bugs.llvm.org/show_bug.cgi?id=25154.
[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17
[4] Adam Chlipala. 2007. A Certified Type-preserving Compiler from Lambda Calculus to Assembly Language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/1250734.1250742
[5] Ștefan Ciobâcă, Dorel Lucanu, Vlad Rusu, and Grigore Roșu. 2014. *A Language-Independent Proof System for Mutual Program Equivalence*. Springer International Publishing, Cham, 75–90. https://doi.org/10.1007/978-3-319-11737-9_6
[6] Clang. 2020. Clang: C Language Family Frontend for LLVM. http://clang.llvm.org. Accessed: February 17, 2021.
[7] Apple Corp. 2019. iOS App Distribution Guide. https://developer.apple.com/library/etc/redirect/DTS/iOSAppDistGuide. Accessed: February 2019.
[8] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. 2020. Scalable Validation of Binary Lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 655–671. https://doi.org/10.1145/3385412.3385964
[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *TACAS'08 (LNCS, Vol. 4963)*. 337–340.
[10] Free Software Foundation. 2019. GNU Compiler Collection Internals. https://gcc.gnu.org/onlinedocs/gccint/Passes.html. Accessed: August 2020.
[11] GCC. 2020. GNU Compiler Collection. https://gcc.gnu.org. Accessed: February 17, 2021.
[12] Jan Friso Groote, David N. Jansen, Jeroen J. A. Keiren, and Anton J. Wijs. 2017. An $O(m\log n)$ Algorithm for Computing Stuttering Equivalence and Branching

Bisimulation. *ACM Trans. Comput. Logic* 18, 2, Article 13 (June 2017), 34 pages. https://doi.org/10.1145/3060140

[13] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will You Still Compile Me Tomorrow? Static Cross-version Compiler Validation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 191–201. https://doi.org/10.1145/2491411.2491442

[14] Jeremy Horwitz. 2018. Apple Watch apps instantly went 64-bit thanks to obscure Bitcode option. *VentureBeat* (2018).

[15] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

[16] Julia. 2020. The Julia Language. http://julialang.org. Accessed: February 17, 2021.

[17] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

[18] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 327–337. https://doi.org/10.1145/1542476.1542513

[19] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[20] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 633–647. https://doi.org/10.1145/3062341.3062343

[21] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[22] LLVM. 2020. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html. Accessed: February 17, 2021.

[23] LLVM. 2020. LLVM Target-Independent Code Generation: Instruction Selection. http://llvm.org/docs/CodeGenerator.html#instruction-selection-section. Accessed: February 17, 2021.

[24] LLVM. 2020. LLVM Target-independent Code Generator. http://llvm.org/docs/CodeGenerator.html#machine-code-representation. Accessed: February 17, 2021.

[25] LLVM. 2020. TableGen. http://llvm.org/docs/TableGen/index.html. Accessed: February 17, 2021.

[26] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

[27] Kedar S. Namjoshi. 1997. *A simple characterization of stuttering bisimulation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 284–296. https://doi.org/10.1007/BFb0058037

[28] Kedar S. Namjoshi and Lenore D. Zuck. 2013. *Witnessing Program Transformations*. Springer Berlin Heidelberg, Berlin, Heidelberg, 304–323. https://doi.org/10.1007/978-3-642-38856-9_17

[29] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

[30] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*. Springer-Verlag, London, UK, UK, 151–166. http://dl.acm.org/citation.cfm?id=646482.691453

[31] Hartley Rogers, Jr. 1987. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA.

[32] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[33] Hanan Samet. 1975. *Automatically Proving the Correctness of Translations Involving Optimized Code*. Ph.D. Dissertation. Stanford, CA, USA. AAI7525601.

[34] Davide Sangiorgi. 2011. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA.

[35] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 471–482. https://doi.org/10.1145/2491956.2462183

[36] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven Equivalence Checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2509136.2509509

[37] SPEC. 2020. SPEC CPU 2006 Benchmark. https://www.spec.org/cpu2006/. Accessed: February 17, 2021.

[38] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbr&#252;cken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

[39] Swift. 2020. Swift programming language. https://swift.org. Accessed: February 17, 2021.

[40] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

[41] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-graph Translation Validation for LLVM. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 295–305. https://doi.org/10.1145/1993498.1993533

[42] David Wheeler. 2015. To Bitcode, or Not to Bitcode? *iovation* (2015).

[43] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-based Optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/2491956.2462164

[44] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. 2003. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science* 9 (2003), 2003.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact accompanying this paper is publicly available in Zenodo: https://doi.org/10.5281/zenodo.4322105. The artifact is packaged as a VirtualBox image. We recommend using VirtualBox version 6.1 or later to import the image and create a virtual machine. The virtual machine contains pre-installed software dependencies, a pre-built set up of our Translation Validation system along with its source code, as well as all the input files required to reproduce the experiments in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Translation Validation of ISel phase of LLVM using Keq
- **Program:** GCC benchmark from SPEC 2006. LLVM Bitcode of GCC functions included.
- **Compilation:** LLVM 5.0.2. K framework with Keq tool. All included.
- **Binary:** LLVM tool binaries, Keq tool. All included.
- **Run-time environment:** Linux (Ubuntu 20.04)
- **Hardware:** Recommended: 16 processing cores at >= 2.67GHz and >= 80GB of memory.
- **How much disk space required (approximately)?:** About 12GB of disk space.
- **How much time is needed to prepare workflow (approximately)?:** Less than 30 mins.
- **How much time is needed to complete experiments (approximately)?:** About 96 hours (with recommended hardware).
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Yes, in Zenodo (DOI 10.5281/zenodo.4322105).

A.3 Description

A.3.1 How to access. Download the VirtualBox image from Zenodo: https://doi.org/10.5281/zenodo.4322105. Import the image to VirtualBox, preferably version 6.1 or later. You can access the artifact from within the created virtual machine. The image (.ova file) is 3.6GB large. The virtual machine requires about 12GB of disk space.

A.3.2 Hardware dependencies. If it is desired to reproduce the full GCC experiment mentioned in the paper, we recommend 16 processing cores at 2.67GHz and a total of 80GB of memory for the host system. This (or better) hardware will allow the completion of the full experiment in about (or less) 96 hours.

A.3.3 Software dependencies. We recommend VirtualBox version 6.1 or later.

A.4 Installation

Open the downloaded VirtualBox image (.ova file) with VirtualBox. If VirtualBox is installed in your system, you can simply double-click the .ova file. An import wizard window will open that allows for editing various settings for the about to be created virtual machine. Please edit the number of CPU cores and amount of RAM memory according to the specifications of your host system. Then click *import* to generate the *llvm-verified-backend* virtual machine. Afterwards, you should be able to see the virtual machine listed in the left side of the main window of the VirtualBox application. To start the virtual machine, select it from the list and click the *Start* button. The virtual machine is an Ubuntu 20.04 Linux system and you will be logged in as the `lvb` user (with password `asplos`).

The virtual machine contains all software dependencies of our Translation Validation system installed. It also contains our system pre-built and ready to be used for experiments. Finally, it contains a README file (found in `/home/lvb/Desktop/README.md`) with instructions on how to run the experiments in the paper as well as individual tests of translation validation with Keq. For completeness, it also contains instructions on how to build and set up the Translation Validation system from its source code, which is also provided in the virtual machine.

A.5 Experiment workflow

We provide a Python script `run-tests.py` (found in `/home/lvb/Desktop/llvm-verified-backend/scripts/`) that is invoked to run experiments with our Translation Validation system. The script is given an LLVM IR function, it then applies the LLVM ISel pass to generate an output Virtual x86 function, it generates a set of synchronization points for the two programs, and it runs Keq with these points to validate the translation. It can be used with different options to run single tests, batches, or the whole GCC experiment mentioned in the paper. The aforementioned README file contains instructions for all the above use cases.

A.6 Evaluation and expected results

This artifact allows the reproduction of the GCC experiment mentioned in the paper. We compile 4732 supported functions of the GCC SPEC 2006 benchmark from LLVM IR to Virtual x86 using the ISel pass of LLVM. We then validate the translations with Keq. In order to reproduce the GCC experiment, the user needs to invoke the Python script `run-tests.py` with a batch of functions that is the whole set of GCC functions supported by our system. The aforementioned README file contains instructions on how to invoke `run-tests.py` to repeat the experiment as well as a recommended set of performance flags. Unfortunately, we cannot be certain that exactly the same number of functions as reported in the paper will be validated, because there may be differences in how many function validations timed out or went out of memory depending on the available hardware resources.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html