Tamarin: Concolic Disequivalence for MIPS

Abel Nieto
University of Waterloo
anietoro@uwaterloo.ca

Abstract. Given two MIPS programs, when are they equivalent? At first glance, this is tricky to define, because of the unstructured nature of assembly code. We propose the use of alternating concolic execution to detect whether two programs are disequivalent. We have implemented our approach in a tool called Tamarin, which includes a MIPS emulator instrumented to record symbolic traces, as well as a concolic execution engine that integrates with the Z3 solver. We show that Tamarin is able to reason about program disequivalence in a number of scenarios, without any a-priori knowledge about the MIPS programs under consideration.

1 Introduction

We are staring at two opaque black boxes laying at our feet. Each box has a narrow slot through which we can place items in the box, but we cannot quite see what is inside. They look approximately like this:

We know each box contains an animal, but we do not know which specific animal is in each one. We would like to find out if both boxes contain the same species of animal. Our solution is simple: we take two carrots, and drop one in each box through the slots.

After a while, a chewing sound emerges from the boxes. We peer into them and, indeed, it looks like the carrots were successfully eaten. Triumphantly, we declare that the boxes contain the same species of animal. The truth is altogether different:

The boxes are assembly programs. The animals are the functions those programs compute. The carrot is unit testing. The task was to determine whether the programs were equivalent. And we failed at it. In this paper, we show a technique that is better than a carrot.

2 Program Equivalence for MIPS

Let us set up the problem a bit more formally. Consider the set $P$ of MIPS-assembly programs that satisfy two restrictions: they take as inputs only the values of registers
$1$ and $2$, and when they stop executing we define their output to be (exclusively) the value of $3$. Other side effects, such as printing values to the screen, or system calls, are disallowed.

We can now define a relation $\text{equiv} \subseteq P \times P$ (and its complement, $\text{equiv}^c$) of equivalent programs. Given $P_1, P_2 \in P$, we say that $P_1 \equiv P_2$ (read “$P_1$ is equivalent to $P_2$”) if, for all inputs $1$ and $2$, one of the following holds:

– Both $P_1$ and $P_2$ fail during execution (for example, due to a divide-by-zero error).
– $P_1$ and $P_2$ stop with the same output in $3$.

For example, the two programs in Figure 1 are equivalent.

![Fig. 1. $P_1 \equiv P_2$](#)

Notice that $P_1 \equiv P_2$ even though $P_2$ modifies the contents of the memory and an additional register ($4$), because:

– Both $P_1$ and $P_2$ terminate without errors.
– The value of $3$ will be the same when they do so.

Unfortunately, even though $\text{equiv}$ captures an already-simplified notion of equivalence\(^1\), a decision procedure for it does not exist, due to Rice’s theorem.

To get decidability back, we define a new class of relations $\text{equiv}_S \subseteq P \times P$ (whose complement is $\text{equiv}^c_S$). We say that $P_1 \equiv_{S} P_2$ (read “$P_1$ is $S$-equivalent to $P_2$”) if, for all inputs, one of the following holds:

– Either $P_1$ or $P_2$ does not stop within $S$ steps (we can think of each CPU cycle as one step).
– Both $P_1$ and $P_2$ fail.
– Both $P_1$ and $P_2$ stop with the same output.

The $\text{equiv}_S$ relation captures the notion that we cannot tell $P_1$ and $P_2$ apart by running them for at most $S$ steps. Figure 2 shows an example of two programs that are $S$-equivalent for $S = 10$, but not equivalent. This is the case because $P_2$ loops while the counter is less than 42, so with 10 steps in our “budget” we will have to stop $P_2$ before the loop is over and we can observe the different result.

Given a fixed $S$, the $\text{equiv}_S$ relation is decidable because there is a finite number of inputs to try, and for each input we only need to run the programs a finite number of steps.

We already saw that equivalence not always implies $S$-equivalence. However, the converse always holds. The following lemma shows that $\text{equiv}_S$ over-approximates $\text{equiv}$.

\(^1\) For example, $\text{equiv}$ has a very narrow notion of output that excludes side effects.
Lemma 1. $\forall S, P_1, P_2, P_1 \equiv_S P_2 \Rightarrow P_1 \equiv_P P_2$.

Proof. Let $P_1 \equiv_P P_2$. Consider two arbitrary inputs $x$ and $y$. Then we have one of two cases:

- Either $P_1$ or $P_2$ (or both) do not stop within $S$ steps when run on $x$ and $y$. This is the first case in the definition of $\equiv_S$.
- Both $P_1$ and $P_2$ stop within $S$ steps. Then because they are equivalent, we know that they either fail with an error, or both stop with the same output. These are the second and third cases in the definition of $\equiv_S$.

Therefore, we must have $P_1 \equiv_S P_2$.

Corollary 1. $P_1 \equiv_S P_2 \Rightarrow P_1 \equiv_P P_2$.

Proof. This is just the contrapositive of Lemma 1.

Corollary 1 can be used to argue the soundness (with respect to $\equiv$) of any decision procedure that under-approximates $\equiv_S$. In the next section we will show one such under-approximation based on concolic execution.

3 Concolic Disequivalence

We know from Corollary 1 that any relation that under-approximates $\equiv_S$ is sound. Figure 3 shows why we want an under-approximation: efficiency. $\equiv$ captures the class of programs that are disequivalent, but is undecidable. $\equiv_S$ is decidable, but likely cannot be computed efficiently. Therefore, we look for a subset of $\equiv_S$ (an under-approximation) that can be efficiently computed.

To fill the missing relation in Figure 3 we propose concolic disequivalence. Abstractly, concolic disequivalence is a function $\text{compare}(P_1, P_2, S)$ that takes as inputs two MIPS programs and returns one of two answers:

- “disequivalent”, in which case $P_1 \equiv_S P_2$.
- “possibly equivalent”, meaning that $P_1$ and $P_2$ might or might not be $S$-equivalent.
Figure 3 shows pseudocode for compare. The algorithm alternately executes $P_1$ and $P_2$. At every step, one of the programs is labelled as the “driver” and the other one as the “verifier”. The driver program is then concolically executed, yielding a set of inputs that exercise a new program path (of the driver). The inputs can then be fed to the verifier, and the results of both driver and verifier compared. If the results are different, then we know $P_1$ and $P_2$ are disequivalent. Otherwise, the driver becomes the verifier, and vice-versa. Eventually, we will traverse all explorable paths, at which point $P_1$ and $P_2$ can be declared possibly equivalent.

We now give an example of how compare operates. Consider the sample programs below:

```plaintext
# P_1
bne $1, 42, end
add $3, $3, $0
end:
add $3, $1, $2

# P_2
add $3, $1, $2
bne $2, 100, end
add $3, $3, $2
end:
```

Figure 5 summarizes the state of the algorithm as it compares $P_1$ and $P_2$. First, notice how the driver and verifier roles flip between $P_1$ and $P_2$ in consecutive runs. Every row indicates the input values, as well as the outputs $R_D$ and $R_V$ of the driver and verifier, respectively. At every run, we also record the path taken by the driver. Path conditions are negated to make sure we explore new paths in every iteration. In the fourth iteration, we can see that compare finds that the input pair $1 = 1, 2 = 100$ leads to different outputs in the driver and verifier. At this point, $P_1$ and $P_2$ are declared as disequivalent.

Notice that in order to uncover the different, it is necessary to concolically explore the paths in both $P_1$ and $P_2$, and not only of $P_1$. In Figure 5, runs 1 and 3 explore both branches of the conditional jump in $P_1$, but they exercise the same path in $P_2$. Only after we also execute $P_2$ do we find a counterexample to equivalence.
function compare($P_1$, $P_2$, $S$)  
$b \leftarrow true$
while either $P_1$ or $P_2$ has unexplored paths do  
if $b$ then  
$D \leftarrow P_1$
$V \leftarrow P_2$
else  
$D \leftarrow P_2$
$V \leftarrow P_1$
end if  
if $D$ has unexplored paths then  
$I \leftarrow$ new inputs that exercise an unexplored path
$R_1 \leftarrow$ run($P_1, I, S$)
$R_2 \leftarrow$ run($P_2, I, S$)
if both $P_1$ and $P_2$ stopped then  
if both $P_1$ and $P_2$ stopped with an error then  
endif
else if either $P_1$ or $P_2$ stopped with an error then  
return “disequivalent”
else  
if $R_1 \neq R_2$ then  
return “disequivalent”
endif
endif
endif  
mark the path discovered by $I$ as explored
$b \leftarrow \neg b$
end if
end while
return “possibly equivalent”
end function

Fig. 4. Concolic disequivalence algorithm

<table>
<thead>
<tr>
<th>Run</th>
<th>Driver</th>
<th>Verifier</th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>Path</th>
<th>$R_D$</th>
<th>$R_V$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$P_1$</td>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
<td>$S_1 \neq 42$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$P_2$</td>
<td>$P_1$</td>
<td>1</td>
<td>1</td>
<td>$S_2 \neq 100$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>$P_1$</td>
<td>$P_2$</td>
<td>42</td>
<td>1</td>
<td>$S_1 = 42$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>$P_2$</td>
<td>$P_1$</td>
<td>1</td>
<td>100</td>
<td>$S_2 = 100$</td>
<td>201</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 5. A sample execution of compare
4 Tamarin

Tamarin\(^2\) is a Scala implementation of the compare algorithm from Section 3. We first give an overview of the major modules in Tamarin, shown in Figure 6, and then describe them in more detail in subsequent sections:

- **Concolic** is the entry point to Tamarin. It implements the top-level loop that visits unexplored paths, alternating between the two programs being compared. Concolic uses the other modules as helpers (in Figure 6, requests made by a module appear as solid lines, and responses to prior requests are shown with dashed lines).
- **CPU** is a MIPS emulator that is instrumented to record symbolic traces containing path conditions, which can later be negated to explore new paths.
- **Trace** consumes raw traces coming from CPU and transforms them in multiple ways so that they can be handed over to the Z3 solver.
- **Query** translates (modified) CPU traces into equivalent logical formulae. The formulae are then solved by the Z3 solver, producing new inputs that, if fed to the program under test, will lead to traversing unexplored paths.
- **Z3** is an SMT solver developed at Microsoft [De Moura and Bjørner, 2008]. We use it as black box for solving queries, over the theories of bitvectors and arrays, that result from program traces.

![Fig. 6. Overview of Tamarin’s modules](image)

4.1 Trace Collection

The CPU module is in charge of running MIPS programs and collecting symbolic traces from executions. It is based on the MIPS emulator written by Ondřej Lhoták for CS241E at the University of Waterloo [Lhotak, ].

The interface to the module consists of a single function:

\(^2\)Tamarins are small-sized monkeys from Central and South America. They are related to marmosets, which are also New World monkeys, and less-importantly give name to the black-box submission and testing server in use at the University of Waterloo as of Fall 2017 [Spacco et al., 2006].
The `run` function takes as input a program represented as a sequence of words, the values of registers $1$ and $2$ (the inputs to the program) and a fuel value (explained below).

The output is an algebraic data type `RunRes` that can take one of three forms:

```scala
```

### Case 1: Done
- If the program executes without error, then `Done(state, trace)` is returned. `state` is the state of the CPU after execution, including the contents of memory (which are ignored) and of register $3$, the output register. `trace` is the symbolic trace captured during the program’s execution, and is described below.

### Case 2: NotDone
- If the program ran for more than `fuel` CPU cycles without stopping, then the result is `NotDone(trace)`. Notice that even though the program did not stop we can still return a trace recording the execution right until the moment we stopped it. The `fuel` argument to `run` plays the same role as the $S$ argument in Figure 3.

### Case 3: Error
- Finally, if there was an error during program execution (for example, an attempted division-by-zero), then we return `Error`. Notice that due to `fuel` parameter and error boxing, the augmented emulator in Tamarin, unlike a vanilla MIPS emulator, is “hardened” in the sense that it can execute MIPS programs that do not stop (the emulator itself will stop) or throw errors (will be caught at the top level by the emulator).

While the emulator is executing a program, it also records a symbolic trace of (most of) the executed instructions. We support a subset of the MIPS instruction set, containing 18 instruction types [Staff, 2017]. Notably, unlike in full MIPS, there are no system calls in our supported subset.

![Fetch Decode Execute MakeSymbolic Trace](image)

**Fig. 7.** Instrumented CPU that records symbolic traces

The fetch-decode-execute cycle in our emulator is modified to record a symbolic representation of each instruction as it is executed (Figure 7). The symbolic instructions are then stored in a trace. There are two types of instructions in a trace: assignments and path conditions.
Assignments are instructions that mutate the CPU state, but do not affect the control flow. The symbolic form of most assignments is very similar to the concrete instruction that is executed. For example, the symbolic representation of the instruction `add $3, $1, $2` is `r₃ ← r₁ + r₂`.

Path conditions are instructions that modify the control flow of the program. This set potentially contains both conditional and unconditional jumps, but we track only conditional ones. Specifically, we record the branch taken at each conditional jump, and symbolically record it as an (in)equality. For example, if the branch `beq $1, $2, foolabel` is not taken, then we will add `r₁ ≠ r₂` to our trace.

One additional point of note: the program counter (`pc`) is not symbolically tracked. This means Tamarin cannot reason about unconditional jumps, so only a single path is explored for the program in Figure 8. There, Tamarin will execute `P` with the initial inputs (a hard-coded constant) and error out, because the jump will lead to an invalid instruction. Tamarin will miss the fact that $1$ and $2$ can be given values such that we could jump to either of the labels, leading to a successful execution.

```plaintext
# P
add $4, $1, $2
jr $4

good:
add $3, $1, $2
jr $31 # $31 contains the termination pc

bad:
add $3, $1, $1
jr $31
```

Fig. 8. Not tracking the pc leads to under-approximations

Since the `pc` is not tracked, whenever it is used in another instruction we need to substitute it by its concrete value. This technique is called concretization [David et al., 2016]. For example, the load immediate and skip instruction has the following semantics:

`lis $d`  
`d ← Mem[pc]; pc ← pc + 4`

Instead of the (precise) interpretation above, we use concretization to generate the (under-approximating) symbolic instruction `d ← v`, where `v` is the value at the address following the `lis` instruction.

### 4.2 Transformations

The `Trace` module also exposes just a single function:

```python
def transform(trace: Trace, depth: Int): Trace
```

`transform` takes as input a trace that was produced by CPU and a `depth` parameter (explained below), and modifies the input trace so that it can be later converted into a logical formula. The `depth` parameter limits the number of path conditions in the outputted trace: this is so we can bound the depth of our DFS as we explore new paths (see Section 4.4 for more details). In effect, we can think of traces as the intermediate
representation (IR) for queries to the SMT solver. The Trace module can be then thought of as lowering the semantic complexity of traces.

Internally, Trace consists of multiple phases, each of type \( \text{Trace} \rightarrow \text{Trace} \), which are sequentially applied to the input. They are described below.

**Desugaring** The desugaring phase removes “complicated” instructions by replacing them with simpler ones. Figure 9 summarizes some of the transformations.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jalr(concretePC)</td>
<td>Add($31, concretePC)</td>
</tr>
<tr>
<td>Mult(s, t)</td>
<td>Mult64(tmp, s, t); Low32(lo, tmp); High32(hi, tmp)</td>
</tr>
<tr>
<td>Div(s, t)</td>
<td>Quot(lo, s, t); Rem(hi, s, t)</td>
</tr>
<tr>
<td>Mflo(d)</td>
<td>Add(d, lo, 0)</td>
</tr>
</tbody>
</table>

Fig. 9. Desugared instructions

A couple of points of interest:

- $31 holds by convention the return address of the current procedure. Since the pc is not tracked, its concretized value is used.
- Notice how some instructions (e.g. Mult) are desugared into multiple instructions: this is because they in fact have more than one side effect in the CPU state.
- While desugaring some instructions, we introduce “helper” instructions like Mult64 and Quot that are not part of the MIPS instruction set. These instructions help communicate the intended semantics to the SMT solver. For example, later on we will see that the contents of registers are represented with 32-bit vectors, but multiplication of two 32-bit integers can take up to 64 bits of storage, so we need a way to let the solver know that it should upcast the product of two registers (so we can later decompose it into the low and high 32-bits): the Mult64 instruction accomplishes this.

**Simplification** The simplification phase removes trivial path conditions, like `beq 0, 0, foolabel`, which might be common (the specific `beq` example is a common idiom for jumping to a loop header) but do not need to be considered by the SMT solver.

**Trimming** The trimming phase just shortens the trace so that it contains at most depth (an argument to Trace) path conditions. This is so we can do a bounded DFS (again, see Section 4.4) to limit the search space.

**SSA Conversion** The SSA phase converts traces into static single assignment (SSA) form [Cytron et al., 1991]. SSA form is crucial to be able to transform a trace into the corresponding logical formula. Consider the example in Figure 10, which shows a small trace together with incorrect and correct translations.

The incorrect translation represents assignment through equality. This is intuitive, but misguided, because assignment mutates state, but equality does not. For example,
suppose we append to our trace the path condition $3 = 9 \land \$1 = 3$. This gets translated to the equality $z = 9 \land x = 3$. But the equalities in the incorrect translation imply that $y = 0$ (because $x = x - y$) and so $z = x = y$. The path condition is then declared as unsatisfiable, but this is clearly wrong, because $\$1 = 6 \land \$2 = 3$ is a solution.

What went wrong, and how do we fix it? As we pointed, assignment mutates the value of registers, so we need to somehow encode that the “$1$” in the last $Add$ instruction is “not the same” as the one in the first $Add$. This behaviour is precisely captured by SSA form, where each variable is assigned to exactly once. This means that two reads of the same register that are separated by a write to the same register will be encoded as different variables. The correct translation in Figure 10 uses subscripts to refer to the “versions” of each variable after writes. If we add the same path condition as before, we can see that it gets translated as $z_1 = 9 \land x_2 = 3$, which is now satisfiable because $x_2 = x_1 - y_1$ does not imply that $x_1 = y_1$.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Incorrect</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Add($3, $1, $2)$</td>
<td>$z = x + y \land z_1 = x_1 + y_1$</td>
<td>$z_1 = x_1 + y_1$</td>
</tr>
<tr>
<td>$Sub($1, $1, $2)$</td>
<td>$x = x - y \land x_2 = x_1 - y_1$</td>
<td>$x_2 = x_1 - y_1$</td>
</tr>
<tr>
<td>$Add($2, $1, $2)$</td>
<td>$y = x + y \land y_2 = x_2 + y_1$</td>
<td>$y_2 = x_2 + y_1$</td>
</tr>
</tbody>
</table>

Fig. 10. A trace together with an incorrect translation to a logical formula and a correct translation using SSA form.

The algorithms for efficiently converting code into SSA form are sophisticated [Cytron et al., 1991], requiring the calculation of so-called dominators and $\phi$-functions. However, we are only interested in converting traces, which do not contain any jumps. In this case, a simple linear-time pass over the trace suffices for converting to SSA. The algorithm is shown in Figure 11.

```
function ssaConvert(trace)
  trace' ← ∅
  ∀x.last(x) ← x₁
  for all instr ∈ trace do
    if instr = Add(d, s, t) then
      s' ← last(s)
      t' ← last(t)
      d₁ ← last(d)
      last(d) ← d₁⁺₁
      instr' ← Add(d₁⁺₁, s', t')
      append(trace', instr')
    else . . . ▷ Other cases
    end if
  end for
  return trace'
end function
```

Fig. 11. SSA conversion of traces
4.3 Queries

The Query module is responsible for interfacing with the Z3 solver to determine the satisfiability of path constraints, leading to unexplored program paths.

The API for the module is the function

```python
def solve(trace: Trace): Option[Soln]
```

The input to `solve` is a trace that has been processed by `Trace`, likely containing a recently-negated path condition (negated by the Concolic module from Section 4.4). `solve`'s task is to convert the trace into an equivalent logical formula, hand it to Z3, and then return the solution found by the solver.

A `Soln` is just a `Seq[RegVal]` (`solve` returns an `Option[Soln]` in case the formula is unsatisfiable), and a `RegVal` is an algebraic datatype of the form

```scala
case object Unbound extends RegVal
case class Fixed(v: Long) extends RegVal
```

A `RegVal` represents the value of a register in the model returned by the solver. `Unbound` represents the case when the solver does not constrain a register in its solution. By contrast, `Fixed` is used when the solver requires a specific value.

**Translation** Figure 12 shows an example of how a program trace can be converted into a logical formula understood by the solver. In this case, the program has a single conditional branch, leading to one path condition. Presumably, we have already explored (perhaps from the initial run with random inputs) what happens when the branch is taken (that is $4 = 5$ and so we skip the last addition). Now we are interested in exploring the case where the branch is not taken: for that, we need the path condition $4 \neq 5$ at the end of the trace.

The third column in Figure 12 shows the logical formula that is produced by the Query module. The formula is satisfiable iff the path condition $4 \neq 5$ can evaluate to true for certain values of $1$ and $2$. In Figure 12, we have used the SMT-LIB [Barrett et al., 2010] representation of the query, which is a textual representation with a Lisp-like syntax. Tamarin uses Z3’s Java API, for efficiency.

Every register in the trace has a corresponding “constant” in the formula ($r_1 \ldots r_6$). The constants have type “32-bit bitvector”, to encode that registers are 32-bit integers. Consequently, arithmetic operations on registers are encoded as operations on bitvectors (e.g., `bvadd`, `bvslt`).

Some instructions like `slt` have domain-specific semantics, so their translations are slightly more complicated: `slt` specifically uses SMT-LIB’s `ite` (if-then-else) construct.

There is no mutation in the formula, but that is ok because of our conversion to SSA from Section 4.2. Instead of mutation, relationships between constants are encoded using assertions. In general, we have one assertion per instruction in the trace.

Once all constant declarations and assertions have been specified, we can query Z3 to check the satisfiability of the formula via `(check-sat)`. If the formula is satisfiable, we can also request the satisfying model that Z3 found, with `get-model`. In this case, Z3 returns

```lisp
(model (define-fun r1 () (_ BitVec 32) #x80000000)
 (define-fun r2 () (_ BitVec 32) #x80000000)
 ...)
```
program:          trace:          SMT-LIB:
add $3, $1, $2   add $3, $1, $2          (declare-const r1 (_ BitVec 32))
slt $4, $1, $2   slt $4, $1, $2          (declare-const r2 (_ BitVec 32))
lis $5           add $5, $1, $0          (declare-const r3 (_ BitVec 32))
1                $4 != $5              (declare-const r4 (_ BitVec 32))
beq $4, $5, skip add $6, $3, $1          (declare-const r5 (_ BitVec 32))

skip:

Fig. 12. A sample program, a trace in it, and the trace’s representation in SMT-LIB notation

Which indeed is a solution (albeit not one a human would have picked), because it makes slt $4, $1, $2 assign 0 to $4, which skips the branch.

Memory Representation In keeping with registers being represented as 32-bit bitvectors, Tamarin represents memory itself as an array of bitvectors. Below we show the translations of the sw and lw instructions:
lw $t, i, $s          (assert (= r_t (select mem_last (bvadd i r_s))))
sw $t, i, $s          (assert (= mem_k+1 (store mem_k (bvadd i r_s) r_t))

Notice how memory is also immutable: every store (sw) operation returns a new memory constant (an array), and every select (lw) uses the latest memory constant.

4.4 Concolic Execution Redux

As mentioned before, the Concolic module implements the concolic execution engine and orchestrates the interactions between all modules. We already presented the general structure in Figure 3, but below we expand on some additional points.

Alternation The main difference between Tamarin and a traditional concolic execution engine is that Tamarin needs to maintain state for the two programs it is comparing, and opposed to just one. Therefore, we represent the engine state with a tuple (trace1, trace2, turn) of the last-executed trace for each program, together with a pointer to the last-run program. At every iteration, we use turn to choose the program that is next in line for execution, and use the corresponding tracei to identify the last path-condition (and the corresponding inputs) that has not been previously
negated. The programs are run on the new inputs, their traces recorded, and a new state \((trace'_1, trace'_2, \text{turn'})\) generated.

**Bounded DFS** As in the original concolic testing tool, CUTE [Sen et al., 2005], Tamarin explores program paths using a bounded depth-first search. To guarantee termination, Tamarin trims CPU traces so that they contain at most depth path conditions (later path conditions are simply ignored). The current depth is set at 50, but is configurable.

**Comparing Results** When are the results of the two programs considered contradictory? Only when both programs terminate with different values in $3$, or when one program fails while the other succeeds. In particular, non-termination is treated conservatively and no inferences are derived from it (this could be modified soundly if one of the two programs is given a “privileged” status by considering it the specification). The full rules are those in Section 2.

**Soundness and Completeness** As we saw in Lemma 1, Tamarin is sound, but not complete. There are multiple sources of incompleteness: not all paths are explored (because of the depth parameter), termination is ensured via the fuel, and part of the program state is simply not tracked at all, leading to under-approximation.

**Efficiency** We have not done a formal complexity analysis of the algorithm implemented by Tamarin. In any case, as with other concolic execution tools, there is a risk of combinatorial path explosion, which can be traded away in exchange for a loss in precision via the depth parameter. A similar tradeoff exists with the fuel parameter. The theories used by Tamarin via Z3 are quantifier-free bitvectors and arrays, including multiplication, which is potentially problematic. If further testing revealed an impact to performance, we would look into concretization strategies to handle the multiplication case.

## 5 Evaluation

To evaluate the correctness of Tamarin, we hand-wrote a set of assembly programs (program pairs, to be more precise) that exercise different functionality in the tool. The test programs are summarized in Figure 13.

<table>
<thead>
<tr>
<th>Program</th>
<th>Tested Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>Can reason about pushes and pops to the stack</td>
</tr>
<tr>
<td>fun</td>
<td>Finds counterexamples across function calls</td>
</tr>
<tr>
<td>while</td>
<td>Able to reason about loops that iterate less than depth times</td>
</tr>
<tr>
<td>infinite</td>
<td>Treats infinite loops conservatively</td>
</tr>
<tr>
<td>div0</td>
<td>Can handle CPU exceptions</td>
</tr>
<tr>
<td>nested</td>
<td>Reasons about nested conditional statements</td>
</tr>
</tbody>
</table>

**Fig. 13.** Test programs for Tamarin
The tests we have to date focus on correctness (and, indirectly, measure efficiency), but they are all small and simple programs. To evaluate whether Tamarin is of practical use, we plan to collect student-written compilers for an undergraduate course at the University of Waterloo. We can then feed these submitted compilers sample programs written in a Scala-like language. We can compare the MIPS code generated by the student’s compilers to that of the reference implementation, using Tamarin. Finally, we will compare Tamarin’s results with the current black-box testing approach used in the course, and investigate whether Tamarin provides better accuracy.

6 Related Work

We now survey some of the existing literature on program equivalence that is relevant to the project. In general, many tools have been built for checking program equivalence, differing on the language they target (high level vs assembly), their level of generality (one-off tools vs frameworks or even intermediate languages that serve as the backend for multiple tools), the language features they support (loops vs loop unrolling) and their soundness and completeness guarantees (concolic testing based tools vs complete exploration of a bounded search space). However, we were not able to find a tool that both does program equivalence and works with raw assembly programs.

The two ecosystems of tools most relevant to our work are KLEE and Boogie.

**KLEE** [Cadar et al., 2008] introduced KLEE, which is now a popular backend for program verification tasks. It implements a symbolic VM for LLVM bitcode. However, unlike Tamarin, the input to KLEE is at the C level. [Ramos and Engler, 2011] build UC-KLEE on top of KLEE. Given two C functions, UC-KLEE checks whether they are equivalent (up to a fixed input size of 8 bytes). Their notion of equivalence considers not only function return values, but also memory locations reachable from them. As far as we can tell, UC-KLEE was never released publicly.

**Boogie** [Barnett et al., 2005] present Boogie, an intermediate verification language that can be targeted as an IR for program verification tasks. [Lahiri et al., 2012] built SymDiff on top of Boogie to test for program equivalence. However, even though Boogie can encode MIPS, SymDiff requires that procedures in the two compared programs be given in pairs via a pre-established correspondence. This is so that Boogie can reason about one procedure at a time, making conservative assumptions about the state of the other procedures. This makes it unclear whether SymDiff could be used to verify equivalence of arbitrary MIPS programs whose structures we do not know a-priori. [Hawblitzel et al., 2013] verify compiler correctness by checking program equivalence of emitted assembly code. They use x86, Boogie, and SymDiff.

**Other related work** [Person et al., 2011] combine static analysis and symbolic execution for program equivalence. [Egele et al., 2014] use Blanket Execution to identify portions of program binaries that are “similar”, based on randomized testing and having similar side-effects. [Partush and Yahav, 2013] use abstract interpretation to prove program equivalence for numerical programs. [Yang et al., 2014] show how verify that manually-annotated program properties continue to hold as a program
evolves. [Necula, 2000] verifies equivalence of IRs before and after compiler passes. [Francesco et al., 2014] present GreASE, which focuses on non-equivalence checking. [Sharma et al., 2013] are able to check equivalence of x86 loops, unlike most of the other tools, which need to unroll loops.

7 Future Work

Tamarin is still very much a work-in-progress. Below we list some of the outstanding work.

Data Structure Generation The CUTE tool [Sen et al., 2005] is able to generate not only numeric inputs, but also C-like structs using a somewhat inductive approach: to generate a struct, first generate null, then generate an instance of the struct where all fields are null, and then iterate the construction. This is harder to do in Tamarin, because it is working at the assembly level. Unlike in C, where we have struct definitions to guide the “shapes” we need to generate, Tamarin has no type-level information to guide the creation of data structures. One option is to specify such shape with metadata that is given to Tamarin as input together with the two programs.

Restarts Because we do not precisely track the semantics of the executed MIPS program (for example, we ignore the pc), it is possible that we solve a set of path constraints, but the resulting input does not lead the MIPS program along the expected path. In these cases, Tamarin currently stops evaluating the “misdirected” program (we will however continue to concolically execute the second program). Another approach, also used in CUTE, is to restart the engine with random inputs. We do not currently know how common restarts are, but they might be worth implementing for more complicated programs.

Experiments As mentioned in Section 5, we would like to run Tamarin with compiler-generated MIPS programs, to test scalability and real-world use. This would be easier to do if we could generate data structures, since many programs take more two integers as input.

8 Conclusions

Program equivalence is a hard problem. Equivalence of assembly language programs is arguably even harder, because it needs to handle the unstructured nature of assembly without type or grammar-level information.

Surprisingly, there does not seem to be a program equivalence tool that works at the assembly level (for arbitrary programs). Tamarin is one step in this direction.
References


