Low-level Reachability Analysis based on Formal Logic

Nico Naus[1,2][0003−3442−1543], Freek Verbeek[1,2][0002−6625−1123], Marc Schoolderman[3][0001−8648−3995], and Binoy Ravindran[1][0000−0002−8663−739X]

[1] Virginia Tech, Blacksburg VA, USA {niconaus,freek,binoy}@vt.edu
[2] Open Univeristy, The Netherlands {niconaus,fvb}@ou.nl
[3] Radboud University Nijmegen, The Netherlands m.schoolderman@cs.ru.nl

Abstract. Reachability is an important problem in program analysis. Automatically being able to show that – and how – a certain state is reachable, can be used to detect bugs and vulnerabilities. Various research has focused on formalizing a program logic that connects preconditions to post-conditions in the context of reachability analysis, e.g., must+, Lisbon Triples, and Outcome Logic. Outcome Logic and its variants can be seen as an adaptation of Hoare Logic and Incorrectness Logic. In this paper, we aim to study 1.) how such a formal reachability logic can be used for automated precondition generation, and 2.) how it can be used to reason over low-level assembly code. Automated precondition generation for reachability logic enables us to find inputs that provably trigger an assertion (i.e., a post-condition). Motivation for focusing on low-level code is that low-level code accurately describes actual program behavior, can be targeted in cases where source code is unavailable, and allows reasoning over low-level properties like return pointer integrity. An implementation has been developed, and the entire system is proven to be sound and complete (the latter only in the absence of unresolved indirections) in the Isabelle/HOL theorem prover. Initial results are obtained on litmus tests and case studies. The results expose limitations: traversal may not terminate, and more scalability would require a compositional approach. However, the results show as well that precondition generation based on low-level reachability logic allows exposing bugs in low-level code.

Keywords: Formal verification · Formal Methods · Reachability analysis

1 Introduction

Reachability is an important problem in program analysis. Being able to automatically show that a certain state is reachable allows us to detect bugs and vulnerabilities in code. Currently most reachability analysis approaches target high-level code [22, 7, 27]. However, high-level code is still an abstraction over the actual program behavior. The actual execution is defined by the assembly code produced by compilation. It is at this level where we can fully reason over properties such as memory safety and return pointer integrity, which are the major cause of software vulnerabilities [23]. For example, an out-of-bounds memory write not prevented by memory unsafe languages such as C can overwrite the return address stored on the stack, before a function returns. Other vulnerabilities and exploitation techniques such as return-oriented-programming (ROP) are also only detectable at the lowest level [5]. An ROP attack is executed by chaining together instructions already present in memory, to perform arbitrary operations. In addition, reasoning over low-level code opens up the ability to reason over reachability in programs where source code is unavailable.

Reasoning over low-level code also has its drawbacks. For starters, memory is not structured, and treated as an array-like structure. As a result, a write or read of a pointer can access any region in memory, go out of bounds or overlap with existing pointers. Control flow is also unstructured. Program execution can jump to any point in the code, and jumps can be dynamic. Resolving these indirections is a challenge. Formal semantics are often unavailable for low-level architectures, requiring any analysis to deal with uncertainty. The aforementioned challenges will be addressed in this paper. Another challenge is obtaining assembly or low-level code from binaries [32]. We consider this out of scope, since this problem is more or less orthogonal.

Logics that reason over reachability triples have been studied extensively under various names (must+ [2, 3], Backwards Under-Approximative Triples [24], Lisbon Triples [26], Outcome Logic [34]). They revolve around triples of the following definition:

$$\langle P \rangle\ p\ \langle Q \rangle \equiv \forall\, \sigma \in \Sigma \cdot P(\sigma) \implies \exists\, \sigma' \in \Sigma \cdot \sigma \xrightarrow{p} \sigma' \wedge Q(\sigma')$$

Here, p is a program under investigation and Q is a postcondition. Intuitively, the triple formulates that any state σ satisfying the precondition P will reach the postcondition with at least one of its execution paths. It is herein different from commonly known program logics such as Hoare Logic [19] which reasons over program correctness, and Reverse Hoare Logic [33] and Incorrectness Logic [26] which both reason over total reachability. A more thorough discussion on the relation between these logics, can be found elsewhere [25, 34]. From now on, we will refer to this kind of triples as reachability triples.

In this paper, we consider postcondition Q to be *fixed* and thus study the problem: can we define a function τ that given program p and postcondition Q computes a precondition such that $\langle \tau(p, Q) \rangle\ p\ \langle Q \rangle$ holds? Moreover, we consider program p to be low-level code.

The relevance of such a function τ, is that it allows the generation of inputs that lead to *unwanted* states. In low-level code, an unwanted state can be a candidate for an exploit. For example, a state in which the top of the stack frame is overwritten is unwanted, as it may lead to an exploit. Taking as postcondition such an unwanted state and applying function τ can either show that the unwanted state is unreachable, or provide information on how to reach it.

The first step is to formalize an academic programming language similar to the well-known WHILE [19] language. Whereas WHILE is intended to be an abstract model of high-level programming languages, this paper proposes JUMP as an abstract model of low-level representations of executable behavior such as assembly or LLVM IR [21]. The language JUMP is characterized by being low-level, having unstructured control flow (jumps instead of loops) and an unstructured flat memory model. Moreover, it is *non-deterministic*, allowing us to model the uncertainty of the semantics of various constructs found in executables. Even state-of-the-art research into semantics of instruction sets are not able to provide deterministic semantics for all instructions [17, 12]. Any static analysis over low-level code thus must be able to deal with the non-determinism caused by undefined behavior of instructions.

We then define a function τ in two forms: 1.) in Isabelle/HOL [25, 19, 11], and 2.) a mirrored implementation in Haskell. The Isabelle/HOL version allows a formal proof of soundness and completeness; we know that the search space describes only actual reachability evidence, and that it describes all possible ways to reach the intended state.

Algorithmically, the approach presented in this paper boils down to backwards symbolic execution (BSE) [10, 9, 13, 15]. What this paper aims to do, is to relate backwards symbolic execution to a program logic, analogous to how forwards symbolic execution is related to Hoare logic. By formulating BSE as a precondition-transformation function over reachability triples, we can formally reason over soundness and completeness.

Limitations of this approach include that the search space becomes infinite. This is a necessary consequence of soundness and completeness. However, finding one path from assertion to initial state suffices and thus there is no need for full search space traversal to find bugs and vulnerabilities. Various research exists that combine BSE with dealing with loops, but the focus of this paper is to show how precondition-generation for reachability logic allows finding unwanted states in low-level code. Additionally, the characterisation of unwanted states, i.e., which postcondition to start with, is now chosen manually. Automating this characterisation is out of scope.

The Haskell implementation allows experimentation on several litmus tests, as well as on two larger case studies. It shows how a search space is generated, traversed and preconditions are found. Application of this approach to large real-world programs is explicitly left as future work. All results, source code and the formalized proof of correctness in Isabelle/HOL are publicly available[4].

In summary, this paper presents the following contributions:

[4] https://github.com/niconaus/low-level-reachability

- A formal foundation for reasoning over reachability in low-level languages.
- A sound and complete precondition-generation algorithm for (single-path) reachability triples.

This paper is the first to provide a formal foundation for reasoning over reachability in low-level languages, in particular, formulate BSE as precondition-transformation function over reachability triples.

Section 2 introduces the JUMP language. Section 3 presents the reachability triples precondition generation mechanism. Litmus tests are described in Section 4 and Section 5 presents two case studies. Related work is discussed in Section 6 before we conclude in Section 7.

2 The JUMP language

The JUMP language is intended as an abstract representation of low-level languages such as LLVM IR [21] or assembly. It has no explicit control flow; instead it has jumps to addresses. It consists of basic *blocks* of elementary statements that end with either a jump or an exit. Blocks are labeled with addresses. Memory is modeled as a mapping from addresses to values. Variables from a set $\mathcal{V}$ represent registers and flags. The *values* stored in variables and memory are words (bit-vectors) of type $\mathcal{W}$.

The following design decisions have been made regarding JUMP.

Non-determinism We explicitly include *non-determinism* through an `Obtain` statement that allows to retrieve some value out of a set. Non-determinism allows modeling of external functions whose behavior is unknown, allows dealing with uncertain semantics of assembly instructions and allows modeling user-input and IO. The `Obtain` statement is the only source of non-determinism in JUMP.

Unstructured memory Memory essentially consists of a flat mapping of addresses to values. There is no explicit notion of heap, stack frame, data section, or global variables. This is purposefully chosen as it allows to reason over pointer aliasing. For example, it allows Reachability Triples to formulate statements as "the initial value of this pointer should be equal to the initial value of register `rsp`" which is interpreted as a pointer pointing to the return address at the top of the stack frame. Note that registers are treated as variables in JUMP.

No structured control flow All control flow happens either through jumps, conditional jumps or indirect jumps. Indirect control flow is typically introduced by a compiler in case of switch statements, callbacks, and to implement dynamic dispatch. Note that a normal instruction such as the x86 instruction `ret` implicitly is an indirect jump as well.

Definition 1. *A* JUMP program p *is defined as the pair* (a_0, blocks) *where* a_0 *is the entry address, and* blocks *a mapping from addresses to blocks. A* block *is defined by the grammar in Figure 1.*

Block

$\underline{\texttt{b}} ::= \underline{\texttt{s}}\texttt{;}\ \underline{\texttt{b}} \mid \texttt{Exit}$	Sequence, exit
$\mid\ \texttt{Jump}\, a \mid \texttt{CJump}\, \underline{\texttt{e}}\ a_1\ a_2 \mid \texttt{IJump}\, \underline{\texttt{e}}$	Jump, conditional jump, indirect jump

Statement

$\underline{\texttt{s}} ::= \texttt{Assign}\, v\ \underline{\texttt{e}}$	Variable assignment
$\mid\ \texttt{Obtain}\, v\ \texttt{Where}\ \underline{\texttt{e}}$	Nondeterministic assign
$\mid\ \texttt{Store}\ \underline{\texttt{e}}_1\ \underline{\texttt{e}}_2$	Store v in address e

Expression

$\underline{\texttt{e}} ::= w \mid v \mid \texttt{*}\underline{\texttt{e}} \mid \underline{\texttt{e}_1} \oplus \underline{\texttt{e}_2} \mid \neg\, \underline{\texttt{e}}$	Value, variable, deref, bin op, not
$\oplus \in \{+, -, \times, \%, <, \leq, =, \neq, >, \geq, \wedge, \vee, \ldots\}$	Binary operators

Fig. 1: The JUMP language syntax.

A block consists of a sequence of zero or more statements, followed by either a jump, conditional jump, indirect jump or `Exit`, where `Exit` merely indicates that a program has ended. The conditional jump jumps to the address a_1 only if the given expression evaluates to non-zero, otherwise to address a_2. The indirect jump calculates the value of $\underline{\texttt{e}}$ and jumps to the block at that address. *Statements* can be assignments or stores. A deterministic assignment writes the value of expression $\underline{\texttt{e}}$ to variable v. A nondeterministic assignment, denoted as $\texttt{Obtain}\, v\ \texttt{Where}\ \underline{\texttt{e}}$, obtains some value w that satisfies $\underline{\texttt{e}}[v := w]$, and writes it to variable v. Note that since expressions can read from memory, using the C-style $*e$ notation, an assignment can model a load instruction. A store writes the value that results from evaluating $\underline{\texttt{e}}_2$ into the memory location that is obtained by evaluating $\underline{\texttt{e}}_1$. *Expressions* consist of values, variables, dereferencing, binary operations and negation.

The state consists of values assigned to variables and memory. Memory is defined as an array-like structure. Two memory operations are provided, namely reading and writing. Function write is of type $\mathcal{A} \times \mathcal{W} \times \mathcal{M} \to \mathcal{M}$ and function read is of type $\mathcal{A} \times \mathcal{M} \to \mathcal{W}$.

We assume values can bijectively be cast to addresses and we do so freely.

Definition 2. *A* state σ *is a tuple* $(\text{mem}, \text{vars})$ *where* mem *is of type* $\mathcal{M}$ *and* vars *are of type* $\mathcal{V} \to \mathcal{W}$.

Semantics are expressed through transition relations $\longrightarrow_{\text{J}}$, $\longrightarrow_{\text{B}}$ and $\longrightarrow_{\text{S}}$ that respectively define state transitions induced by programs, blocks, and statements (see Figure 2). For example, notation $p : \ \sigma \longrightarrow_{\text{J}} \sigma'$ denotes a transition induced by program p from state σ to state σ'. Notation $\sigma \vdash \underline{\texttt{e}} = w$ denotes the evaluation of expression $\underline{\texttt{e}}$ in state σ to value w.

The semantics are largely straightforward. A program defined by an entry address a_0 and a mapping blocks from addresses to basic blocks, is evaluated by evaluating the block pointed to by the entry address. A conditional jump is evaluated by evaluating the condition, and then the target block. Indirect jumps are evaluated in a similar manner, by evaluating the expression to obtain the

$$\textsc{Prog}\ \frac{\mathrm{blocks}(a_0) \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma'}{(a_0, \mathrm{blocks}) \ :\ \sigma \longrightarrow_{\mathrm{J}} \sigma'} \qquad \textsc{Seq}\ \frac{\underline{\mathtt{s}} \ :\ \sigma \longrightarrow_{\mathrm{S}} \sigma' \qquad \underline{\mathtt{b}} \ :\ \sigma' \longrightarrow_{\mathrm{B}} \sigma''}{\underline{\mathtt{s}};\underline{\mathtt{b}} \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma''} \qquad \textsc{Exit}\ \frac{}{\mathtt{Exit} \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma}$$

$$\textsc{Jump}\ \frac{\mathrm{blocks}(a) \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma'}{\mathtt{Jump}\, a \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma'} \qquad \textsc{IJump}\ \frac{\sigma \vdash \underline{\mathtt{e}} = a \qquad \mathrm{blocks}(a) \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma'}{\mathtt{IJump}\, \underline{\mathtt{e}} \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma'}$$

$$\textsc{CJumpLeft}\ \frac{\sigma \vdash \underline{\mathtt{e}} \neq 0 \qquad \mathrm{blocks}(a_1) \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma'}{\mathtt{CJump}\, \underline{\mathtt{e}}\ a_1\ a_2 \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma'} \qquad \textsc{CJumpRight}\ \frac{\sigma \vdash \underline{\mathtt{e}} = 0 \qquad \mathrm{blocks}(a_2) \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma'}{\mathtt{CJump}\, \underline{\mathtt{e}}\ a_1\ a_2 \ :\ \sigma \longrightarrow_{\mathrm{B}} \sigma'}$$

$$\textsc{Assign}\ \frac{(\mathrm{mem}, \mathrm{vars}) \vdash \underline{\mathtt{e}} = w}{\mathtt{Assign}\, v\ \underline{\mathtt{e}} \ :\ (\mathrm{mem}, \mathrm{vars}) \longrightarrow_{\mathrm{S}} (\mathrm{mem}, \mathrm{vars}[v/w])}$$

$$\textsc{Store}\ \frac{(\mathrm{mem}, \mathrm{vars}) \vdash \underline{\mathtt{e}_1} = a \qquad (\mathrm{mem}, \mathrm{vars}) \vdash \underline{\mathtt{e}_2} = w}{\mathtt{Store}\, \underline{\mathtt{e}_1}\ \underline{\mathtt{e}_2} \ :\ (\mathrm{mem}, \mathrm{vars}) \longrightarrow_{\mathrm{S}} (\mathsf{write}(a, w, \mathrm{mem}), \mathrm{vars})}$$

$$\textsc{NDAssign}\ \frac{(\mathrm{mem}, \mathrm{vars}) \vdash \underline{\mathtt{e}}[v/w] \neq 0}{\mathtt{Obtain}\, v\ \mathtt{Where}\ \underline{\mathtt{e}} \ :\ (\mathrm{mem}, \mathrm{vars}) \longrightarrow_{\mathrm{S}} (\mathrm{mem}, \mathrm{vars}[v/w])} \qquad \textsc{Load}\ \frac{(\mathrm{mem}, \mathrm{vars}) \vdash \underline{\mathtt{e}} = a \qquad \mathsf{read}(a, \mathrm{mem}) = w}{(\mathrm{mem}, \mathrm{vars}) \vdash *\underline{\mathtt{e}} = w}$$

Fig. 2: Semantics of JUMP. Rules for evaluation of expressions are omitted, except for the dereference operator.

block to jump to. The nondeterministic assignment NDASSIGN is non-standard, and evaluates expression $\underline{\mathtt{e}}$ after substituting the variable v for some value w. For any value w where expression $\underline{\mathtt{e}}$ evaluates to non-zero, a transition may occur. A STORE evaluates expression $\underline{\mathtt{e}_1}$ producing some address a, and evaluates $\underline{\mathtt{e}_2}$ and writes its value to the corresponding region in memory. A LOAD uses function read to read from memory. All other expression evaluations are omitted because they are standard.

3 Precondition generation

Precondition generation has its basis in reachability triples, as defined in Section 1. Motivation for choosing reachability triples as our underlying logic, is that it is the only program logic triple that is suitable for generating inputs automatically. Hoare Logic [19] requires reasoning over all paths, and thus needs manually written loop invariants. Reverse Hoare Logic [33] and Incorrectness Logic [26] allow for an over-approximation of the set of input states, leading to false positives. For a more in-depth discussion on the differences between these logics, and the advantage of using reachability triples, we refer to work by Zilberstein et al. [34].

Using the reachability triple definition from Section 1, we can now define our precondition generation function. The central idea is to formulate a transforma-

tion function τ that takes as input 1.) a program p, and 2.) a post-condition Q, and produces as output a disjunctive *set* of preconditions. This transformation function follows the recursive structure of JUMP, i.e., we formulate functions τ_J, τ_B and τ_S that perform transformations relative to a program, a block and a statement respectively.

Predicate
$P ::= \exists\, i \in \mathbb{N} \cdot \underline{\mathsf{e}} \wedge P \mid \underline{\mathsf{e}}$ Existential quantification, expression

Predicates P are expressions (true if and only if they evaluate to non-zero), but can also contain outermost existential quantifiers. The predicate $\exists i \in \underline{\mathsf{e}} \cdot P$ means there exists a value w for i such that both $\underline{\mathsf{e}}[i/w]$ and $P[i/w]$ hold.

When applied statement-by-statement, the τ-functions populate the precondition search space. This search space is an acyclic graph, with symbolic predicates as vertices and the initial postcondition as the root. It contains a labeled edge (Q, s, P) if and only if application of function τ_S for statement s and postcondition Q produces a set containing precondition P.

Given a program p and a postcondition Q defined in the predicate language above, a transformation is *sound* if it generates preconditions P that form a reachability triple. Soundness means that a generated precondition actually represents an initial state that non-deterministically leads to the Q-state. To define soundness, we first define the notion of a reachability triple relative to blocks, instead of a whole program as in Section 1:

Definition 3. *A reachability triple for block b is defined as:*

$$\langle P \rangle\ b\ \langle Q \rangle \equiv \forall\, \sigma \in \Sigma \cdot P(\sigma) \implies \exists\, \sigma' \cdot b : \ \sigma \longrightarrow_{\mathrm{B}} \sigma' \wedge Q(\sigma')$$

We restate this definition to stress that a reachability triple over block b intuitively means that precondition P leads to the desired state when running the block *and subsequent blocks jumped to, until an exit*, i.e, not just running the instructions within block b itself. This is due to the nature of the transition relation $\longrightarrow_{\mathrm{B}}$ (see Figure 2). A similar definition can also be made for statements: a reachability triple $\langle P \rangle\ s\ \langle Q \rangle$ for statement s is defined for transition relation $\longrightarrow_{\mathrm{S}}$ and thus concerns the execution of the individual statement s only.

Definition 4. *Function τ_J is* sound*, if and only if, for any program p and postcondition Q:*

$$\forall P \in \tau_J(p, Q) \cdot \langle P \rangle\ p\ \langle Q \rangle$$

Similarly, soundness is defined for blocks and statements, with the only difference that the precondition for blocks and statements is constructed by combining the predicate and path condition in conjunction.

Figure 3 shows the transformation functions. Function τ_P starts at the entry block of the program. The program is then traversed in the style of a *right fold* [30]: starting at the entry block, the program is traversed up to an exit point, from which postcondition transformation happens. Function τ_B is identical to standard weakest precondition generation in the cases of sequence and exit. In

Program:

$\tau_J(p, Q) = \tau_B(\text{blocks}(a_0), Q)$

Block:

$\tau_B(\underline{\texttt{s}}\texttt{;}\underline{\texttt{b}}, Q) = \bigcup\{\tau_S(\underline{\texttt{s}}, P) \mid P \in \tau_B(\underline{\texttt{b}}, Q)\}$

$\tau_B(\texttt{Jump}\,\underline{\texttt{e}}\ a_1\ a_2, Q) = \{P_1 \wedge \underline{\texttt{e}} \mid P_1 \in \tau_b(\text{blocks}(a_1), Q)\}$
$\cup\{P_2 \wedge \neg\underline{\texttt{e}} \mid P_2 \in \tau_b(\text{blocks}(a_2), Q)\}$

$\tau_B(\texttt{IJump}\,\underline{\texttt{e}}, Q) = \{P \wedge \underline{\texttt{e}} \equiv a \mid P \in \tau_b(\text{blocks}(a), Q), a \in \text{dom}(\text{blocks})\}$

$\tau_B(\texttt{Exit}, Q) = \{Q\}$

Statement:

$\tau_S(\texttt{Assign}\ v\ \underline{\texttt{e}}, Q) = \{Q[v/\underline{\texttt{e}}]\}$

$\tau_S(\texttt{Obtain}\ v\ \texttt{Where}\,\underline{\texttt{e}}, Q) = \{\exists i \in \underline{\texttt{e}} \cdot Q[v/i]\}$

$\tau_S(\texttt{Store}\ \underline{\texttt{e}}\ v, Q) = \{Q' \wedge \phi \mid (Q', \phi) \in \tau_{\texttt{store}}(\underline{\texttt{e}}, v, Q)\}$

Fig. 3: Precondition generation functions

the case of a conditional jump, two paths are explored. Either path could lead to a precondition, as long as the branching conditions remain satisfiable. In case of an indirect jump, all possible addresses that can be jumped to, are explored.

Function τ_S is standard in case of deterministic assignment. In case of nondeterministic assignment, according to the execution semantics, some value i needs to be found that fulfills the condition $\underline{\texttt{e}}$. That existentially quantified value is substituted for variable v in the post-condition.

In the case of memory assignment, predicate transformation is a bit more complex. Consider the following example:

$$\texttt{Store}\ x\ 42\texttt{; Store}\ y\ 43\ \langle *x = 42\rangle$$

If memory regions x and y alias, then $*x$ will be 43 after execution. The postcondition $*x \equiv 42$ can only hold if x and y are separate.

We explicitly encode assumptions about memory separation into the generated preconditions. The $\tau_{\texttt{store}}$ function listed in Figure 4 takes care of this. It takes as input expression $\underline{\texttt{e}}_1$ that describes a memory pointer, expression $\underline{\texttt{e}}_2$ which is the value to be written, and the postcondition P. It returns a set of tuples (Q, ϕ) where Q is the precondition and ϕ provides the pointer-relations under which that substitution holds. For example, we have $\tau_{\texttt{store}}(a_1, v, *a_2 = 42) = \{(*a_2 = 42, a_1 \neq a_2), (v = 42, a_1 = a_2)\}$. This indicates two possible substitutions when transforming postcondition into precondition:

$$\langle *a_2 = 42\rangle\ \texttt{Store}\ a_1\ v\ \langle *a_2 = 42\rangle\ \text{ if }\ a_1 \neq a_2$$
$$\langle v = 42\rangle\ \ \texttt{Store}\ a_1\ v\ \langle *a_2 = 42\rangle\ \text{ if }\ a_1 = a_2$$

All other cases of $\tau_{\texttt{store}}$ merely propagate the case generation.

There are no special rules for dealing with loops. Instead, loops are unrolled by the precondition generation. In the case of infinite iterations, the reachability search space will be infinitely large. To deal with this search space, we order and prune the space. Theorem 1 states a basic property of reachability triples that

$$
\begin{aligned}
\tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, w) &= \{(c,\ \mathit{True})\} \\
\tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, v) &= \{(x,\ \mathit{True})\} \\
\tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, *\underline{\mathtt{e}}_p) &= \{(*\underline{\mathtt{e}}_p, \underline{\mathtt{e}}_1 \neq \underline{\mathtt{e}}_p), (\underline{\mathtt{e}}_2, \underline{\mathtt{e}}_1 = \underline{\mathtt{e}}_p)\} \\
\tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, \neg\, \underline{\mathtt{e}}_p) &= \{(\neg\, \underline{\mathtt{e}}'_p, \phi) \mid (\underline{\mathtt{e}}'_p, \phi) \in \tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, \underline{\mathtt{e}}_p)\} \\
\tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, \underline{\mathtt{e}}_{p1} \oplus \underline{\mathtt{e}}_{p2}) &= \{(\underline{\mathtt{e}}'_{p1} \oplus \underline{\mathtt{e}}'_{p2}, \phi_1 \wedge \phi_2) \mid (\underline{\mathtt{e}}'_{p1}, \phi_1) \in \tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, \underline{\mathtt{e}}_{p1}) \\
&\qquad , (\underline{\mathtt{e}}'_{p2}, \phi_2) \in \tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, \underline{\mathtt{e}}_{p2})\} \\
\tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, \exists i \in\ \underline{\mathtt{e}}_p \cdot P) &= \{(\exists i \in \underline{\mathtt{e}}'_p \cdot P', \phi_1 \wedge \phi_2) \mid (\underline{\mathtt{e}}'_p, \phi_1) \in \tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, \underline{\mathtt{e}}_p) \\
&\qquad , (P', \phi_2) \in \tau_{\texttt{store}}(\underline{\mathtt{e}}_1, \underline{\mathtt{e}}_2, P)\}
\end{aligned}
$$

Fig. 4: Case definitions for precondition of store

is used for the purpose of pruning. Section 4 describes how the space is ordered to manage large search spaces.

Theorem 1 (Preservation of unsatisfiability). *For any program p and conditions P and Q such that* $\langle P \rangle\ p\ \langle Q \rangle$,

$$(\forall\, \sigma' \cdot Q(\sigma') \implies \mathit{False}) \implies (\forall\, \sigma \cdot P(\sigma) \implies \mathit{False})$$

The above can directly be concluded from the definition of a reachability triple, as given at the beginning of this section. Once an unsatisfiable condition is generated, the precondition generation can be halted, and the condition discarded.

We validate our precondition generation function by proving it is both sound and complete. Theorems 2 and 3 define these respective properties.

Theorem 2 (Soundness of precondition generation). *Functions* τ_P, τ_B *and* τ_S *are sound.*

Theorem 3 (Completeness of precondition generation).

$$\frac{\mathit{termination(p,P)} \qquad \mathit{no\_indirections(p)}}{\langle P \rangle\ p\ \langle Q \rangle \implies \exists P' \in \tau_J(p, Q) \wedge (P \implies P')}$$

Having both soundness and completeness means that the reachability space defines all and only valid preconditions for a certain program and postcondition.

Both theorems, including 1.) the syntax and semantics of JUMP, 2.) the syntax and semantics of the predicates, and 3.) the functions τ have been formally proven correct in the Isabelle/HOL theorem prover. The proof, including a small example of precondition generation within Isabelle/HOL, constitutes roughly 1000 lines of code. Proof scripts are publicly available[5]. To prove completeness, Theorem 3 imposes two restrictions. One, we require execution of a program p under a state described by P to terminate. If a program does not terminate, it is impossible to construct a P' for this program, and therefore completeness does not hold. Two, we show the theorem holds for programs without indirect

[5] https://github.com/niconaus/low-level-reachability

jumps. The predicate no_indirections(p) ensures that the JUMP program does not contain an `IJump` instruction. In practice however, this premise has little to no impact. Every JUMP program containing indirect jumps, can be converted to one with only direct jumps, by encoding a jump-table like structure using blocks and conditional jumps. Given that P is a precondition for program p and post-condition Q, the precondition generation will generate a P' that is non-strictly weaker than P. An equivalent of Theorem 3 also holds for τ_S and τ_B.

4 Litmus tests

This section presents two litmus tests that demonstrate the application of reachability triples and its precondition generation algorithm to low-level code. We have a prototype implementation available in Haskell, in which we have tested these examples[6].

The prototype implements the τ functions similar to how they are presented above. The τ functions are defined as non-deterministic functions, building up a tree as a search space. Branches at the same level originate from a conditional, and deeper branches indicate a jump. On top of that, basic simplification is applied to the generated predicates, to make them more readable.

The precondition search space can be infinitely large. The implementation builds up the search space as a tree structure. This orders the search space, making it feasible to search the infinite space in a structured way. Although some rudimentary ordering is done, efficiently searching and reducing the reachability space is explicitly left as future work. The implementation includes an SMT solver, for deciding the satisfiability of the computed preconditions.

4.1 Infinite reachability space: Long division

Our first litmus test demonstrates conditional jumps, loops, infinite reachability space and post-condition pruning. Figure 5 lists the program blocks on the left. The blocks are labeled #0 though #3, with block #0 the entry point. Variables x and y signify the input. The program divides x by y, by means of long division. If x is larger than y, the result of the division is returned in variable i. The variable x is updated, and after execution holds the remainder from division.

In this case, we want to derive that a state is reachable which clearly should not be, to show that there is a bug in the program. The program behaves incorrectly when after execution, the remainder stored in x is equal to or larger than the divisor y. We use this, $x \geq y$, as our postcondition.

The right side of Figure 5 represents precondition generation. Conditions shown in this Figure are left unsimplified for the purpose of illustration. We start back to front. `Exit` does not alter the postcondition, so we just copy it. Then, we either execute block 0, 1 or 2, depending on what condition holds. If we came directly from block 0, then $x < y$ must hold, so our precondition is

[6] https://github.com/niconaus/low-level-reachability

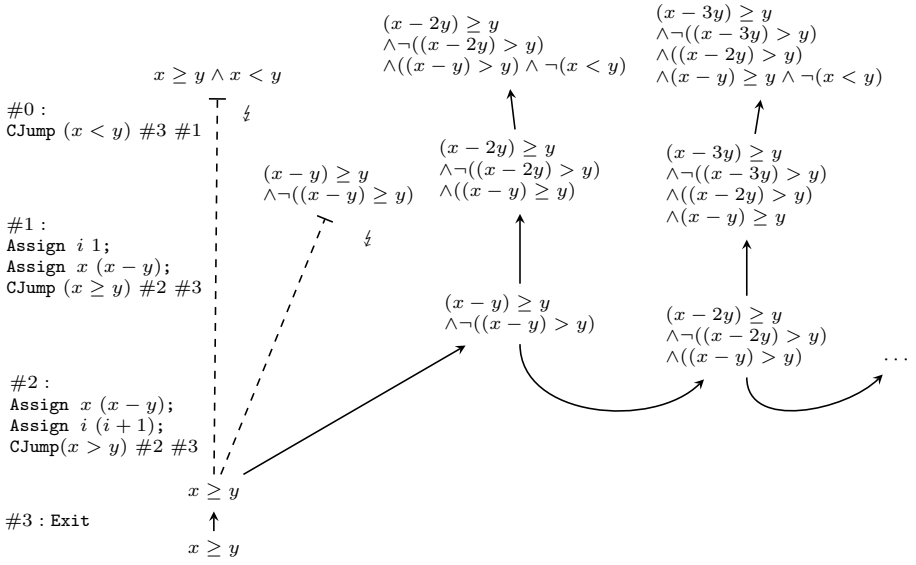

Fig. 5: Precondition generation for long division example. A dashed arrow leads to an unsatisfiable precondition.

$x \geq y \wedge x < y$, which is false, indicated by the lightning bolt. If we came from block 1, then $\neg(x \geq y)$ must have held. Block 1 updates x with $x - y$, leading to the precondition $(x - y) \geq y \wedge \neg((x - y) \geq y)$. Note that this precondition is unsatisfiable. From Theorem 1, we know that we can halt exploration of this particular path.

The last block to look at, is block 2. To arrive here, we must have had that $\neg(x > y)$. The body of block 2 updates x, and we end up with $(x - y) \geq y \wedge \neg((x - y) > y)$. Here, we see the loop unfolding at work. We have executed the loop body once, and the τ function generates two alternatives. We exit the loop, indicated by the arrow pointing up, or we run another iteration, indicated by the arrow pointing right.

Ending the loop at this point again leads to a precondition that is satisfiable. Completing the calculation, leads us to the first viable precondition for the post-condition $x \geq y$.

The precondition function τ does not stop at this point. A second unrolling step is shown in the Figure. It will continue to unroll the loop an infinite amount of times, making the reachability space infinitely large. By ordering the space as shown in this example, we can perform a breadth first search, starting with the smallest number of unrolling. While this does make the space more manageable, the search space is still potentially infinite. In such a case, if no satisfiable precondition exists, breadth first search will never terminate.

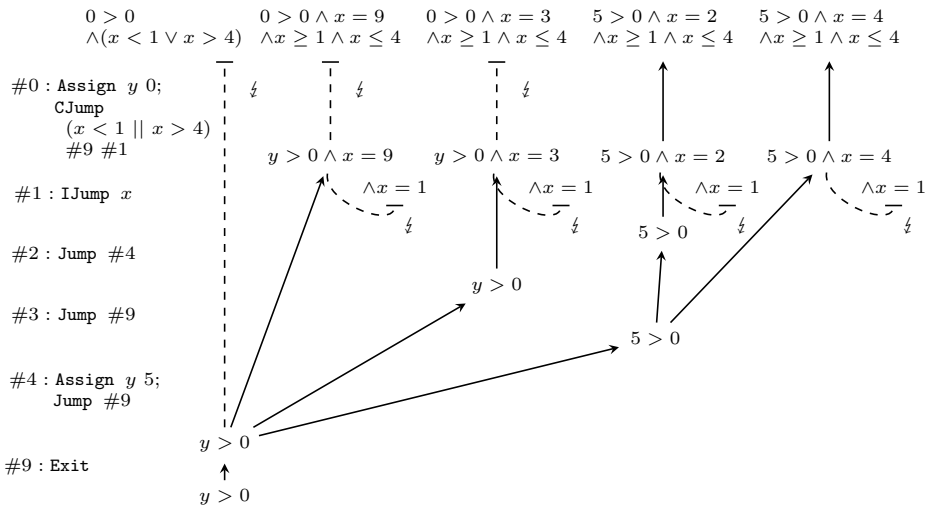

Fig. 6: Precondition generation for indirect jump example.

4.2 Indirect Jumps

Our next litmus test demonstrates how reachability triples and its precondition generation deals with indirect jumps. Switch-statements consisting of many cases are often compiled into jump tables. These are typically combined with a guard for values not handled by the jump table. Figure 6 shows a model of this.

Execution starts at block 0. Here, y is set to 0, and the conditional jump checks if x is smaller than 1 or larger than 4. If so, we jump to exit. If not, we jump to block 1, which is the start of our guard. The indirect jump jumps to the block label stored in x. Blocks 2, 3, and 4 signify the guard options.

As a postcondition, we select $y > 0$. This postcondition does not necessarily encode a program error, but does allow us to demonstrate how our approach deals with indirection.

Starting at block 9, we again work our way up the execution back to front. We refrain from a step-by-step explanation of the precondition generation, and instead focus on the behavior of precondition generation involving indirection. Block 1 contains the indirect jump. As can be seen from the precondition generation graph, we have to explore every possible jump target when we get to block 1, including a jump to itself. This generates a large number of paths, but many of these explorations generate unsatisfiable preconditions. Potentially, an indirect jump can jump to any address, but in practice, the number of paths explored is limited by the conditions that must hold.

5 Case Studies

In this section, we present our results of applying precondition generation to two bigger examples. Out goal with these examples is to demonstrate the feasibility of our approach, but we leave application to real-world binaries as future work.

5.1 Faulty Partitioning for Quicksort

The core of any quicksort algorithm is the partitioning algorithm. One well-known partitioning algorithm is the one invented by Tony Hoare [18] which selects a *pivot* element and then transforms an input data set into two smaller sets, depending on relative ordering of elements in the data set to the pivot. This scheme seems superficially very simple, but it is very easy to get wrong. For instance, the following algorithm has a superficially plausible variant of this partitioning scheme, which is "nearly correct".

```
void quicksort(int a[], size_t N) {
  if(N <= 1) return;
  int pivot = a[rand()%N];
  int i = 0, j = N-1;
  while(i <= j) {
    while(i < j  && a[i] <= pivot) i++;
    while(i <= j && a[j] >= pivot) j--;
    swap(&a[i++], &a[j--]);
  }
  quicksort(a, j+1);
  quicksort(a+i, N-i);}
```

The partitioning scheme can be translated into a JUMP program relatively easily; selection of the pivot can be modeled using a non-deterministic assign.

We are interested in detecting out-of-bounds memory access. We add bounds checks to the program, and thus our postcondition is $(i < 0) \vee (i \geq N)$. Running the resultant program through our implementation for an array of size 3 will then generate an exploit-precondition: the program can go out of bounds if the following condition holds:

$$\exists i. 0 \leq i \leq 2 \wedge a[i] \leq a[0] \wedge a[i] \leq a[1] \wedge a[i] \leq a[2] \wedge a[0] > a[i]$$

Informally, this conditions says that `a[0]` is not the minimal element of the array. The reason for this is that if the minimal element is chosen as a pivot, and `a[0]` is not equal to it, the first inner loop will simply fall through, and after the second loop, i will become -1, pointing outside the array before the swap occurs. A fix for this would be make the swap conditional, replacing it with:

```
if(i <= j) swap(&a[i++], &a[j--]);
```

This will in fact prevent any out-of-bound memory access. However, another way any version of quicksort can fail dramatically is when the recursive calls are performed with incorrect parameters. For example if $i = 0$ or $j = N$ at the end of the partitioning scheme, we will end up in a infinite recursive loop. If we specify this as a post-condition of the partitioning scheme, we find that the same preconditions are generated as before.

The functional correctness of the partitioning scheme can also be examined—that is, is it actually the case that all the elements moved towards the the left-hand side of the array are less-or-equal to the pivot, and that the elements to the right are greater-or-equal than the pivot? To examine this, we can specify as an exploit condition that the input to the first recursive invocation of `quicksort` contains an element greater than the pivot; this finds no satisfiable conditions (as it is not true). However, specifying this for input sent to the second invocation of `quicksort` instead, our prototype will essentially start generating counter-examples. For example, if the first element is the pivot, and strictly less than the middle element but strictly higher than the third element, partitioning fails.

5.2 Karatsuba

Several assembly routines for multiplying multi-precision integers on an 8-bit AVR controller were verified by Schoolderman [29]. It was discovered that some of these routines could compute incorrect results if their arguments aliased with the memory location intended to store the result. A full verification like this appears to require significant effort; however, if we are only interested in finding aliasing bugs, reachability triples seem ideally suited to find these.

We focused on the smallest routine exhibiting the problem: the $48 \times 48 \rightarrow$ 96-bit multiplication routine as originally developed by Hutter and Schwabe [20]. This routine computes a product of two 48-bit integers using Karatsuba's method, splitting its inputs into two 24-bit halves, and performing a three $24 \rightarrow$ 48-bit multiplications with these, combining the results.[7] In the process, the lowest 24-bits of the result are known early on and written to memory before the upper half of the inputs is read, causing an aliasing bug.

To model this in JUMP, registers and the carry flags are modeled as JUMP variables, whereas the memory space is modeled using JUMP addresses. Every AVR instruction is modeled by a sequence of JUMP statements. For example, the instruction ADD $a0$, $a1$ can be expressed by the sequence:

$$\begin{aligned}&\texttt{Assign } tmp\ (a0 + a1);\\ &\texttt{Assign } a0\ (tmp \bmod 256);\\ &\texttt{Assign } carry\ (tmp\ /\ 256)\end{aligned}$$

Adding the appropriate binary operators to the syntax of Figure 1, every instruction required for the program (which are only a handful) can be modeled,

[7] To be more precise, this method uses the fact that $(2^w X_h + X_l)(2^w Y_h + Y_l) = (1 + 2^w)(2^w X_h Y_h + X_l Y_l) - 2^w (X_l - X_h)(Y_l - Y_h)$

allowing the entire multiplication routine (consisting of 136 instructions) to be expressed as a JUMP program. The memory accesses, which operate on three bytes at a time, were modeled as a single memory operations on a three-byte memory region.

As seen in Section 4, generated preconditions can be fairly verbose, and we expected that in this case as well. To remedy this somewhat, we extended the Haskell implementation with constant folding and other simplifications to more efficiently manage the search space of possible preconditions, and pruning areas of the search space which can easily be determined to be impossible. In a more production-oriented setting, SMT solving and/or a robust expression simplifier can be used to do this more efficiently than our naive Haskell implementation.

For the precondition, we look at the case $X \cdot Y$ where $X = Y = 2^{24}$. Clearly the expected result should be $X \cdot Y = 2^{48}$, i.e. the 96-bit result should consist of 12 bytes, all of which contain 0, except for the seventh byte which should hold 1. As a postcondition, we therefore specify that this byte does *not* hold 1.

Running the JUMP version of the 48-bit Karatsuba code through our analysis resulted in a handful of preconditions. Some of these simplify to *False*, as they express impossible aliasing conditions—an SMT solver would be able to discard these easily. However, 7 preconditions remained which are completely plausible and satisfiable, which fall into three categories:

- X, Y alias, and their high 24-bits overlap with the low 24-bits of the result
- X, Y are disjoint, and of them partially overlaps with the result as before
- X, Z are partially aliased, and one of them partially overlaps with the result

Which are exactly the case we would expect: the issue is being caused by either (or both) inputs sharing their high 24-bits with the low 24-bits of the output location. Had we not chosen the fixed input values for X and Y, this case would have generated more complex preconditions, however, this case shows that there is an easy instance where these would be satisfied.

6 Related work

As mentioned in Section 1, the relation described by reachability triples has been studied before under different names. Möller, O'Hearn and Hoare [24] describe what they call Backwards Under-Approximative Triples. They do not develop a precondition generation algorithm for these triples, but merely reflect on the triples with regards to over-approximate triples.

The must+ relation used in works by Ball (et al.) [2, 3] also describes an under-approximative transition relation in the context of abstract interpretation. A must+ transition is defined such that if an abstract transition exists, given a concrete state that relates to the abstract state before execution, a concrete post-state also exists. Instead of doing precondition generation, their aim is to use this relation in a model transition system, to ultimately generate test cases that cover the entire reachable state space.

Zilberstein et al. [34] refer to this exact same relation as Outcome Logic (OL). They argue why OL is better suited for reasoning over reachability, compared to existing program logics. Deduction rules are presented, and the paper includes several example proofs over Outcome Logic triples.

To the best of our knowledge, these logics and triples have never been applied to automatically reason over reachability in low-level code.

Many other approaches to do reachability analysis exist. Dynamic logic [16] allows reasoning over the execution of a program. Use of modal operators $\Box$ and $\Diamond$ allows for reasoning that something is necessarily the case or possibly the case, respectively. For example, stating $\langle p \rangle a$ means that after performing program p, it is possible for a to hold. Reachability triples go beyond this by including the state before execution in the relation, crucial for the intended purpose, as well as defining precondition generation.

Rosu et al. [28] and the continuation of that work by Ştefănescu et al. [31] introduce Reachability Logic for non-deterministic languages. Their logic serves as a proof system that allows for user-assisted reachability proofs over programs. Reachability Logic is language agnostic, and has its basis in Hoare Logic. As mentioned in Section 2, Hoare Logic is unsuitable for automatically generating reachability evidence, and these arguments also extend to Reachability Logic.

Recent work by Asadi et al. [1] describes an under-approximative reachability analysis for linear and polynomial systems. They define the reachability problem as a finite system of linear inequalities and use Farkas' lemma [14] to solve it. Their approach is able to handle theoretical benchmarks that were previously beyond reach. For the purpose of automatic reachability analysis for low-level programs however, their system is too restrictive.

Symbolic execution is another popular method for reasoning over reachability. Symbolic execution runs a program with symbols instead of actual input. Running the program with these symbolic inputs results in a complete overview of the programs behavior. Symbolic execution is extensively used for software testing [4, 6, 7]. Cadar and Sen [8] provide a great overview of the applications of symbolic execution for this purpose. The biggest downside of symbolic execution is that it describes the complete program behavior, and therefore quickly becomes infeasible, due to the many paths to be described.

Symbolic backward execution (SBE) attempts to mitigate the downside of reasoning over all possible paths by targeting a specific program point. Charreteur and Gotlieb present a method for generating test input based on SBE for Java bytecode [11]. Dinges and Agha augment this approach with concrete execution as well [13]. As mentioned earlier, SBE relates to reachability triples, as Hoare logic relates to forward symbolic execution. SBE provides a concrete algorithm, and potential optimizations such as loop invariant generation, to compute preconditions. Our work provides a formal foundation for reasoning over reachability in low level languages, as opposed to a purely algorithmical solution.

7 Conclusion

In this paper, we have studied automated reachability analysis over low-level code based on formal logic. We define low-level code as code with unstructured control flow, unstructured memory model and non-determinism. The use of formal logic based on reachability triples allows us to prove that generated preconditions will lead to a certain post-condition. The formal logic is based on reachability triples, which under various names have been studied earlier in related work [2, 3, 24, 26, 34]).

The precondition generation that has currently been implemented is relatively naive, and may get stuck in an infinite search. In order to apply this kind of reasoning to real-world programs, we believe that further research on efficient ways of traversing the program and compositional reasoning are needed.

Acknowledgements This work is supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4028, by DARPA under Agreement No. HR00112090028, and by the US Office of Naval Research (ONR) under grants N00014-17-1-2297 and N00014-18-1-2665.

References

1. Asadi, A., Chatterjee, K., Fu, H., Goharshady, A.K., Mahdavi, M.: Polynomial reachability witnesses via stellensätze. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 772–787. ACM (2021)
2. Ball, T.: A theory of predicate-complete test coverage and generation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, Third International Symposium, FMCO 2004, Leiden, The Netherlands, November 2 - 5, 2004, Revised Lectures. Lecture Notes in Computer Science, vol. 3657, pp. 1–22. Springer (2004)
3. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for falsification. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3576, pp. 67–81. Springer (2005)
4. Boyer, R.S., Elspas, B., Levitt, K.N.: Select—a formal system for testing and debugging programs by symbolic execution. ACM SigPlan Notices **10**(6), 234–245 (1975)
5. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to RISC. In: Ning, P., Syverson, P.F., Jha, S. (eds.) Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008. pp. 27–38. ACM (2008)
6. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 1020 states and beyond. Information and computation **98**(2), 142–170 (1992)

7. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
8. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
9. Chalupa, M., Strejcek, J.: Backward symbolic execution with loop folding. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12913, pp. 49–76. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_3
10. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 363–374 (2009)
11. Charreteur, F., Gotlieb, A.: Constraint-based test input generation for java bytecode. In: IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010. pp. 131–140. IEEE Computer Society (2010)
12. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86-64 user-level instruction set architecture. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1133–1148 (2019)
13. Dinges, P., Agha, G.A.: Targeted test input generation using symbolic-concrete backward execution. In: Crnkovic, I., Chechik, M., Grünbacher, P. (eds.) ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. pp. 31–36. ACM (2014)
14. Farkas, J.: Theorie der einfachen ungleichungen. Journal für die reine und angewandte Mathematik (Crelles Journal) **1902**(124), 1–27 (1902)
15. Gulwani, S., Juvekar, S.: Bound analysis using backward symbolic execution. Technical Report MSR-TR-2004–95, Microsoft Research (2009)
16. Harel, D.: Dynamic logic. In: Handbook of philosophical logic, pp. 497–604. Springer (1984)
17. Heule, S., Schkufza, E., Sharma, R., Aiken, A.: Stratified synthesis: automatically learning the x86-64 instruction set. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 237–250 (2016)
18. Hoare, C.A.R.: Algorithm 64: Quicksort. Commun. ACM **4**(7), 321 (Jul 1961)
19. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
20. Hutter, M., Schwabe, P.: Multiprecision multiplication on AVR revisited. Journal of Cryptographic Engineering **5**(3), 201–214 (2015)
21. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88 (2004)
22. Le, Q.L., Raad, A., Villard, J., Berdine, J., Dreyer, D., O'Hearn, P.W.: Finding real bugs in big programs with incorrectness logic. Proceedings of the ACM on Programming Languages **6**(OOPSLA1), 1–27 (2022)
23. Miller, M.: Trends, challenge, and shifts in software vulnerability mitigation, 2019. URL https://github.com/microsoft/MSRC-Security-Research/raw/master/presentations/2019_02_BlueHatIL/2019_01 (2019)

24. Möller, B., O'Hearn, P.W., Hoare, T.: On algebra of program correctness and incorrectness. In: Fahrenberg, U., Gehrke, M., Santocanale, L., Winter, M. (eds.) Relational and Algebraic Methods in Computer Science - 19th International Conference, RAMiCS 2021, Marseille, France, November 2-5, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13027, pp. 325–343. Springer (2021)
25. Naus, N., Verbeek, F., Schoolderman, M., Ravindran, B.: Reachability logic for low-level programs. CoRR **abs/2204.00076** (2022)
26. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL), 10:1–10:32 (2020)
27. Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. pp. 225–252 (2020)
28. Rosu, G., Stefanescu, A., Ciobâcă, Ş., Moore, B.M.: One-path reachability logic. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013. pp. 358–367. IEEE Computer Society (2013)
29. Schoolderman, M.: Verifying branch-free assembly code in why3. In: Paskevich, A., Wies, T. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 66–83. Springer International Publishing, Cham (2017)
30. Sheard, T., Fegaras, L.: A fold for all seasons. In: Proceedings of the conference on Functional programming languages and computer architecture. pp. 233–242 (1993)
31. Stefanescu, A., Ciobâcă, Ş., Mereuta, R., Moore, B.M., Serbanuta, T., Rosu, G.: All-path reachability logic. In: Dowek, G. (ed.) Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8560, pp. 425–440. Springer (2014)
32. Verbeek, F., Bockenek, J.A., Fu, Z., Ravindran, B.: Formally verified lifting of c-compiled x86-64 binaries. In: Jhala, R., Dillig, I. (eds.) PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. pp. 934–949. ACM (2022)
33. de Vries, E., Koutavas, V.: Reverse hoare logic. In: Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings. pp. 155–171 (2011)
34. Zilberstein, N., Dreyer, D., Silva, A.: Outcome logic: A unifying foundation for correctness and incorrectness reasoning. CoRR **abs/2303.03111** (2023)