Symbolic Liveness Analysis of Real-World Software

Daniel Schemmel[1], Julian Büning[1], Oscar Soria Dustmann[1], Thomas Noll[2], and Klaus Wehrle[1]

[1] Communication and Distributed Systems, RWTH Aachen University, Germany
{schemmel,buening,soriadustmann,wehrle}@comsys.rwth-aachen.de
[2] Software Modeling and Verification, RWTH Aachen University, Germany
noll@cs.rwth-aachen.de

Abstract. Liveness violation bugs are notoriously hard to detect, especially due to the difficulty inherent in applying formal methods to real-world programs. We present a generic and practically useful liveness property which defines a program as being live as long as it will eventually either consume more input or terminate. We show that this property naturally maps to many different kinds of real-world programs.
To demonstrate the usefulness of our liveness property, we also present an algorithm that can be efficiently implemented to dynamically find lassos in the target program's state space during Symbolic Execution. This extends Symbolic Execution, a well known dynamic testing technique, to find a new class of program defects, namely liveness violations, while only incurring a small runtime and memory overhead, as evidenced by our evaluation. The implementation of our method found a total of five previously undiscovered software defects in BusyBox and the GNU Coreutils. All five defects have been confirmed and fixed by the respective maintainers after shipping for years, most of them well over a decade.

Keywords: Liveness Analysis, Symbolic Execution, Software Testing, Non-Termination Bugs

1 Introduction

Advances in formal testing and verification methods, such as Symbolic Execution (SymEx) [10–12, 22–24, 42, 49] and Model Checking [5, 6, 13, 17, 21, 27, 29, 30, 43, 50], have enabled the practical analysis of real-world software. Many of these approaches are based on the formal specification of temporal system properties using sets of infinite sequences of states [1], which can be classified as either safety, liveness, or properties that are neither [31]. (However, every linear-time property can be represented as the conjunction of a safety and a liveness property.) This distinction is motivated by the different techniques employed for proving or disproving such properties. In practical applications, safety properties are prevalent. They constrain the finite behavior of a system, ensuring that "nothing bad" happens, and can therefore be checked by reachability analysis. Hence, efficient algorithms and tools have been devised for checking such properties that return a finite counterexample in case of a violation [34].

Liveness properties, on the other hand, do not rule out any finite behavior but constrain infinite behavior to eventually do "something good" [2]. Their checking generally requires more sophisticated algorithms since they must be able to generate (finite representations of) infinite counterexamples. Moreover, common finite-state abstractions that are often employed for checking safety do generally not preserve liveness properties.

While it may be easy to create a domain-specific liveness property (e.g., "a `GET / HTTP/1.1` must eventually be answered with an `HTTP/1.1 {status}`"), it is much harder to formulate *general* liveness properties. We tackle this challenge by proposing a liveness property based on the notion of programs as implementations of algorithms that transform input into output:

Def. 1. *A program is* live *iff it always eventually consumes input or terminates.*

By relying on input instead of output as the measure of progress, we circumnavigate difficulties caused by many common programming patterns such as printing status messages or logging the current state.
Detection. We present an algorithm to detect violations of this liveness property based on a straightforward idea: Execute the program and check after each instruction if the whole program state has been encountered before (identical contents of all registers and addressable memory). If a repetition is found that does not consume input, it is deterministic and will keep recurring ad infinitum. To facilitate checking real-world programs, we perform the search for such *lassos* in the program's state space while executing it symbolically.
Examples. Some examples that show the generality of this liveness property are: 1. Programs that operate on input from files and streams, such as `cat`, `sha256sum` or `tail`. This kind of program is intended to continue running as long as input is available. In some cases this input may be infinite (e.g., `cat -`). 2. Reactive programs, such as `calc.exe` or `nginx` wait for events to occur. Once an event occurs, a burst of activity computes an answer, before the software goes back to waiting for the next event. Often, an event can be sent to signal a termination request. Such events are input just as much as the contents of a file are.

In rare cases, a program can intuitively be considered live without satisfying our liveness property. Most prominent is the `yes` utility, which will loop forever, only printing output. According to our experience the set of useful programs that intentionally allow for an infinite trace consuming only finite input is very small and the violation of our liveness property can, in such cases, easily be recognized as intentional. Our evaluation supports this claim (cf. Sec. 6).
Bugs and Violations. The implementation of our algorithm detected a total of five unintended and previously unknown liveness violations in the GNU Coreutils and BusyBox, all of which have been in the respective codebases for at least 7 to 19 years. All five bugs have been confirmed and fixed within days. The three implementations of `yes` we tested as part of our evaluation, were correctly

detected to not be live. We also automatically generated liveness violating input programs for all `sed` interpreters.

1.1 Key Contributions

This paper presents four key contributions:

1. The definition of a generic liveness property for real-world software.
2. An algorithm to detect its violations.
3. An open-source implementation of the algorithm, available on GitHub[3], implemented as an extension to the Symbolic Execution engine KLEE [10].
4. An evaluation of the above implementation on a total of 354 tools from the GNU Coreutils, BusyBox and toybox, which so far detects five previously unknown defects in widely deployed real-world software.

1.2 Structure

We discuss related work (Sec. 2), before formally defining our liveness property (Sec. 3). Then, we describe the lasso detection algorithm (Sec. 4), demonstrate the practical applicability by implementing the algorithm for the SymEx engine KLEE (Sec. 5) and evaluate it on three real-world software suites (Sec. 6). We finally discuss the practical limitations (Sec. 7) and conclude (Sec. 8).

2 Related Work

General liveness properties [2] can be verified by *proof-based methods* [40], which generally require heavy user support. Contrarily, our work is based upon the state-exploration approach to verification. Another prominent approach to verify the correctness of a system with respect to its specification is automatic *Model Checking* using automata or tableau based methods [5].

In order to combat state-space explosion, many optimization techniques have been developed. Most of these, however, are only applicable to safety properties. For example, *Bounded Model Checking (BMC)* of software is a well-established method for detecting bugs and runtime errors [7,18,19] that is implemented by a number of tools [16, 38]. These tools investigate finite paths in programs by bounding the number of loop iterations and the depth of function calls, which is not necessarily suited to detect the sort of liveness violations we aim to discover. There is work trying to establish completeness thresholds of BMC for (safety and) liveness properties [33], but these are useful only for comparatively small systems. Moreover, most BMC techniques are based on boolean SAT, instead of SMT, as required for dealing with the intricacies of real-world software.

Termination is closely related to liveness in our sense, and has been intensively studied. It boils down to showing the well-foundedness of the program's transition relation by identifying an appropriate ranking function. In recent works, this is

[3] https://github.com/COMSYS/SymbolicLivenessAnalysis

accomplished by first synthesizing conditional termination proofs for program fragments such as loops, and then combining sub-proofs using a transformation that isolates program states for which termination has not been proven yet [8]. A common assumption in this setting is that program variables are mathematical integers, which eases reasoning but is generally unsound. A notable exception is AProVE [28], an automated tool for termination and complexity analysis that takes (amongst others) LLVM intermediate code and builds a SymEx graph that combines SymEx and state-space abstraction, covering both byte-accurate pointer arithmetic and bit-precise modeling of integers. However, advanced liveness properties, floating point values, complex data structures and recursive procedures are unsupported. While a termination proof is a witness for our liveness property, an infinite program execution constitutes neither witness nor violation. Therefore, *non-termination* proof generators, such as TNT [26], while still related, are not relevant to our liveness property.

The authors of Bolt [32] present an entirely different approach, by proposing an in-vivo analysis and correction method. Bolt does not aim to prove that a system terminates or not, but rather provides a means to force already running binaries out of a long-running or infinite loop. To this end, Bolt can attach to an unprepared, running program and will detect loops through memory snapshotting, comparing snapshots to a list of previous snapshots. A user may then choose to forcefully break the loop by applying one of two strategies as a last-resort option. Previous research into in-vivo analysis of hanging systems attempts to prove that a given process has run into an infinite loop [9]. Similarly to Bolt, Looper also attaches to a binary but then uses Concolic Execution (ConEx) to gain insight into the remaining, possible memory changes for the process. This allows for a diagnosis of whether the process is still making progress and will eventually terminate. Both approaches are primarily aimed at understanding or handling an apparent hang, not for proactively searching for unknown defects.

In [35], the authors argue that non-termination has been researched significantly less than termination. Similar to [14,25], they employ static analysis to find every Strongly Connected SubGraph (SCSG) in the Control Flow Graph (CFG) of a given program. Here, a Max-SMT solver is used to synthesize a formulaic representation of each node, which is both a quasi-invariant (i.e., always holding after it held once) and edge-closing (i.e., not allowing a transition that leaves the node's SCSG to be taken). If the solver succeeds for each node in a reachable SCSG, a non-terminating path has been found.

In summary, the applicability of efficient methods for checking liveness in our setting is hampered by restrictions arising from the programming model, the supported properties (e.g., only termination), scalability issues, missing support for non-terminating behavior or false positives due to over-approximation. In the following, we present our own solution to liveness checking of real-world software.

3 Liveness

We begin by formally defining our liveness property following the approach by Alpern and Schneider [1–3], which relies on the view that liveness properties do not constrain the finite behaviors but introduce conditions on infinite behaviors. Here, possible behaviors are given by (edge-labeled) transition systems.

Def. 2 (Transition System). *A* transition system *T is a tuple $(S, Act, \rightarrow, I)$:*
- *S is a finite set of states,*
- *Act is a finite set of actions,*
- *$\rightarrow \subseteq S \times Act \times S$ is a transition relation (written $s \xrightarrow{\alpha} s'$), and*
- *$I \subseteq S$ is the set of initial states.*

For $s \in S$, the sets of outgoing actions *is denoted by $Out(s) = \{\alpha \in Act \mid s \xrightarrow{\alpha} s' \text{ for some } s' \in S\}$. Moreover, we require T to be* deadlock free*, i.e., $Out(s) \neq \emptyset$ for each $s \in S$. A* terminal state *is indicated by a self-loop involving the distinguished action $\downarrow \in Act$: if $\downarrow \in Out(s)$, then $Out(s) = \{\downarrow\}$.*

The self-loops ensure that all *executions* of a program are infinite, which is necessary as terminal states indicate successful completion in our setting.

Def. 3 (Executions and Traces). *An (infinite)* execution *is a sequence of the form $s_0\alpha_1 s_1 \alpha_2 s_2 \ldots$ such that $s_0 \in I$ and $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for every $i \in \mathbb{N}$. Its* trace *is given by $\alpha_1\alpha_2 \ldots \in Act^\omega$.*

Def. 4 (Liveness Properties).
- *A* linear-time property *over Act is a subset of Act^ω.*
- *Let $\Pi \subseteq Act$ be a set of* productive actions *such that $\downarrow \in \Pi$. The Π-*liveness property *is given by $\{\alpha_1\alpha_2 \ldots \in Act^\omega \mid \alpha_i \in \Pi$ for infinitely many $i \in \mathbb{N}\}$.*

A liveness property is generally characterized by the requirement that each finite trace prefix can be extended to an infinite trace that satisfies this property. In our setting, this means that in each state of a given program it is guaranteed that eventually a productive action will be performed. That is, infinitely many productive actions will occur during each execution. As $\downarrow$ is considered productive, terminating computations are live. This differs from the classical setting where terminal states are usually considered as deadlocks that violate liveness.

We assume that the target machine is deterministic w.r.t. its computations and model the consumption of input as the only source of non-determinism. This means that if the execution is in a state in which the program will execute a non-input instruction, only a single outgoing (unproductive) transition exists. If the program is to consume input on the other hand, a (productive) transition exists for every possible value of input. We only consider functions that provide at least one bit of input as input functions, which makes $\downarrow$ the only productive action that is also deterministic, that is, the only productive transition which must be taken once the state it originates from is reached. More formally, $|Out(s)| > 1 \Leftrightarrow Out(s) \subseteq \Pi \setminus \{\downarrow\}$. Thus if a (sub-)execution $s_i\alpha_{i+1}s_{i+1}\ldots$ contains no productive transitions beyond $\downarrow$, it is fully specified by its first state s_i, as there will only ever be a single transition to be taken.

Similarly, we assume that the target machine has finite memory. This implies that the number of possible states is finite: $|S| \in \mathbb{N}$. Although we model each possible input with its own transition, input words are finite too, therefore Act is finite and hence $Out(s)$ for each $s \in S$.

4 Finding Lassos

Any trace t that violates a liveness property must necessarily consist of a finite prefix p that leads to some state $s \in S$, after which no further productive transitions are taken. Therefore, t can be written as $t = pq$, where p is finite and may contain productive actions, while q is infinite and does not contain productive actions. Since S is a finite set and every state from s onward will only have a single outgoing transition and successor, q must contain a cycle that repeats itself infinitely often. Therefore, q in turn can be written as $q = fc^{\omega}$ where f is finite and c non-empty. Due to its shape, we call this a *lasso* with pf the *stem* and c the *loop*.

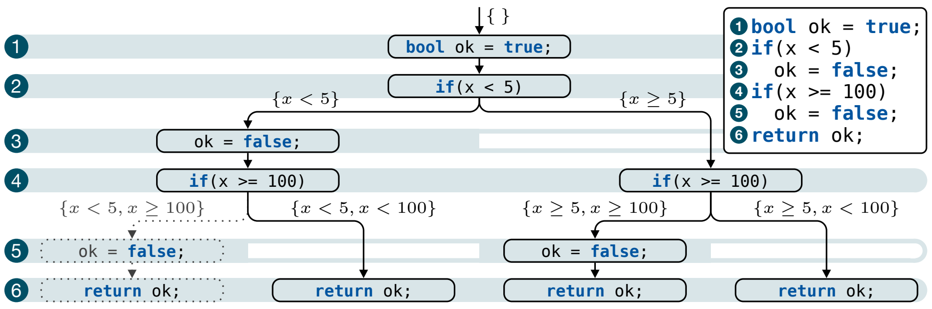
Due to the infeasible computational complexity of checking our liveness property statically (in the absence of input functions, it becomes the finite-space halting problem), we leverage a dynamic analysis that is capable of finding any violation in bounded time and works incrementally to report violations as they are encountered. We do so by searching the state space for a lasso, whose loop does not contain any productive transitions. This is naïvely achieved in the dynamic analysis by checking whether any other state visited since the last productive transition is equal to the current one. In this case the current state deterministically transitions to itself, i.e., is part of the loop.

To implement this idea without prohibitively expensive resource usage, two main challenges must be overcome: 1. Exhaustive exploration of all possible inputs is infeasible for nontrivial cases. 2. Comparing states requires up to 2^{64} byte comparisons on a 64 bit computer. In the rest of this section, we discuss how to leverage SymEx to tackle the first problem (Sec. 4.1) and how to cheapen state comparisons with specially composed hash-based fingerprints (Sec. 4.2).

4.1 Symbolic Execution

Symbolic Execution (SymEx) has become a popular dynamic analysis technique whose primary domain is automated test case generation and bug detection [10–12,15,22,41,42,49]. The primary intent behind SymEx is to improve upon exhaustive testing by symbolically constraining inputs instead of iterating over all possible values, which makes it a natural fit.

Background. The example in Fig. 1 tests whether the variable `x` is in the range from 5 to 99 by performing two tests before returning the result. As `x` is the input to this snippet, it is initially assigned an unconstrained symbolic value. Upon branching on `x < 5` in line 2, the SymEx engine needs to consider two cases: One in which `x` is now constrained to be smaller than `5` and another one in which it is constrained to *not* be smaller than `5`. On the path on which `x <`



```
1 bool ok = true;
2 if(x < 5)
3   ok = false;
4 if(x >= 100)
5   ok = false;
6 return ok;
```

Fig. 1: SymEx tree showing the execution of a snippet with two `if`s. The variable `x` is symbolic and one state is unreachable, as its Path Constraint is unsatisfiable.

5 held, `ok` is then assigned **`false`**, while the other path does not execute that instruction. Afterwards, both paths encounter the branch `if (x >= 100)` in line 4. Since the constraint set $\{x < 5, x \geq 100\}$ is unsatisfiable, the leftmost of the four resulting possibilities is unreachable and therefore not explored. The three remaining paths reach the return statement in line 6. We call the set of currently active constraints the *Path Constraint (PC)*. The PC is usually constructed in such a way, as to contain constraints in the combined theories of quantifier-free bit-vectors, finite arrays and floating point numbers[4].

Symbolic Execution of the Abstract Transition System. By using symbolic values, a single SymEx state can represent a large number of states in the transition system. We require that the SymEx engine, as is commonly done, never assigns a symbolic value (with more than one satisfying model) to the instruction pointer. Since the productive transitions of the transition system are derived from instructions in the program code, this means that each instruction that the SymEx engine performs either corresponds to a number of productive, input-consuming transitions, or a number of unproductive, *not* input-consuming transitions. Therefore, any lasso in the SymEx of the program is also a lasso in the transition system (the $\downarrow$ transition requires trivial special treatment).

To ensure that the opposite is also true, a simple and common optimization must be implemented in the SymEx engine: Only add branch conditions to the PC that are not already implied by it. This is the case iff exactly one of the two branching possibilities is satisfiable, which the SymEx engine (or rather its SMT solver) needs to check in any case. Thereby it is guaranteed that if the SymEx state is part of a loop in the transition system, not just the concrete values, but also the symbolic values will eventually converge towards a steady state. Again excluding trivial special treatment for program termination, a lasso in the transition system thus entails a lasso in the SymEx of the program.

[4] While current SymEx engines and SMT solvers still struggle with the floating point theory in practice [37], the SMT problem is decidable for this combination of theories. Bitblasting [20] gives a polynomial-time reduction to the boolean SAT problem.

4.2 Fingerprinting

To reduce the cost of each individual comparison between two states, we take an idea from hash maps by computing a *fingerprint* ρ for each state and comparing those. A further significant improvement is possible by using a strong cryptographic hash algorithm to compute the fingerprint: Being able to rely (with very high probability) on the fingerprint comparison reduces the memory requirements, as it becomes unnecessary to store a list of full predecessor states. Instead, only the fingerprints of the predecessors need to be kept.

Recomputing the fingerprint after each instruction would still require a full scan over the whole state at each instruction however. Instead, we enable efficient, incremental computation of the fingerprint by not hashing everything, but rather hashing many small *fragments*, and then composing the resulting hashes using bitwise xor. Then, if an instruction attempts to modify a fragment f, it is easy to compute the old and new fragment hashes. The new fingerprint ρ_{new} can then be computed as $\rho_{\text{new}} \coloneqq \rho_{\text{old}} \oplus \mathit{hash}(f_{\text{old}}) \oplus \mathit{hash}(f_{\text{new}})$. Changing a single fragment therefore requires only two computations and bitwise xors on constant size bit strings—one to remove the old fragment from the composite and one to insert the new one. Each incremental fingerprint update only modifies a small number of fragments statically bounded by the types used in the program.

4.3 Algorithm Overview

The proposed algorithm explores as much of the input state as is possible within a specified amount of time, using SymEx to cover large portions of the input space simultaneously. Every SymEx state is efficiently checked against all its predecessors by comparing their fingerprints.

5 Efficient Implementation of the Algorithm

To develop the algorithm presented in the previous section into a practically useful program, we decided to build upon the KLEE SymEx engine [10], with which many safety bugs in real-world programs have been previously found [10,15,41]. As KLEE in turn builds upon the LLVM compiler infrastructure [36], this section begins with a short introduction to LLVM Intermediate Representation (IR) (Sec. 5.1), before explaining how the fragments whose hashes make up the fingerprint can be implemented (Sec. 5.2) and how to track fingerprints (Sec. 5.3). Finally, we detail a technique to avoid as many comparisons as possible (Sec. 5.4).

5.1 LLVM Intermediate Representation

LLVM Intermediate Representation (IR) was designed as a typed, low-level language independent from both (high-level) source language and any specific target architecture, to facilitate compiler optimizations. It operates on an unlimited number of typed registers of arbitrary size, as well as addressable memory. Instructions in IR operate in Static Single Assignment (SSA) form, i.e., registers

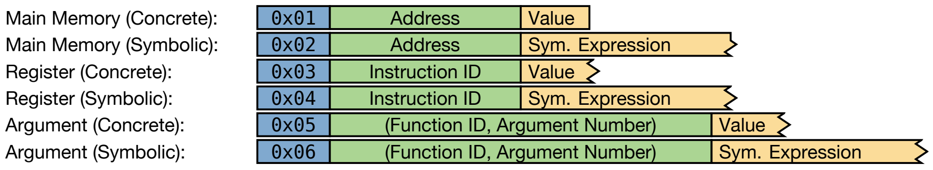


Fig. 2: Six kinds of fragments suffice to denote all possible variants. Symbolic values are written as serialized symbolic expressions consisting of all relevant constraints. All other fields only ever contain concrete values, which are simply used verbatim. Fields of dynamic size are denoted by a ragged right edge.

are only ever assigned once and never modified. The language also has functions, which have a return type and an arbitrary number of typed parameters. Apart from global scope, there is only function scope, but IR features no block scope.

Addressable objects are either global variables, or explicitly allocated, e.g., using `malloc` (cleaned up with `free`) or `alloca` (cleaned up on return from function).

5.2 Fragments

When determining what is to become a fragment, i.e., an atomic portion of a fingerprint, two major design goals should be taken into consideration:

1. Collisions between hashed fragments should not occur, as they would expunge one another from the fingerprint. This goal can be decomposed further:
   (a) The hashing algorithm should be chosen in a manner that makes collisions so unlikely, as to be non-existent in practice.
   (b) The fragments themselves need to be generated in a way that ensures that no two different fragments have the same representation, as that would of course cause their hashes to be equal as well.
2. Fragment sizes should be as close as possible to what will be modified by the program in one step. Longer fragments are more expensive to compute and hash, and shorter fragments become invalidated more frequently.

Avoiding Collisions. In order to minimize the risk of accidental collisions, which would reduce the efficacy of our methodology, we chose the cryptographically secure checksum algorithm BLAKE2b [4] to generate 256 bit hashes, providing 128 bit collision resistance. To the best of our knowledge, there are currently no relevant structural attacks on BLAKE2b, which allows us to assume that the collision resistance is given. For comparison: The revision control system GIT currently uses 160 bit SHA-1 hashes to create unique identifiers for its objects, with plans underway to migrate to a stronger 256 bit hash algorithm[5].

To ensure that the fragments themselves are generated in a collision-free manner, we structure them with three fields each, as can be seen in Fig. 2. The first field contains a tag that lets us distinguish between different types of

[5] https://www.kernel.org/pub/software/scm/git/docs/technical/hash-function-transition.html (Retrieved Jan. 2018)

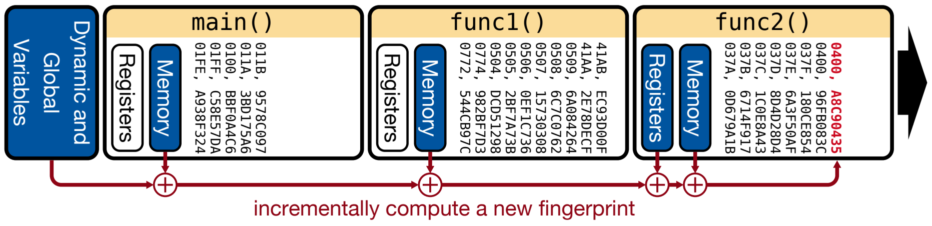

Dynamic and Global Variables
main()
Registers
Memory
func1()
Registers
Memory
func2()
Registers
Memory
incrementally compute a new fingerprint

Fig. 3: Incremental computation of a new fingerprint. Fingerprints are stored in a call stack, with each stack frame containing a partial fingerprint of all addressable memory allocated locally in that function, another partial fingerprint of all registers used in the function and a list of previously encountered fingerprints. A partial fingerprint of all dynamic and global variables is stored independently.

fragments, the middle field contains an address appropriate for that type, and the last field is the value that the fragment represents. We distinguish between three different address spaces: 1. main memory, 2. LLVM registers, which similarly to actual processors hold values that do not have a main memory address, and 3. function arguments, which behave similarly to ordinary LLVM registers, but require a certain amount of special handling in our implementation. For example, the fragment (`0x01`, `0xFF3780`, `0xFF`) means that the memory address `0xFF3780` holds the concrete byte `0xFF`. This fragment hashes to `ea58`...`f677`.

If the fragment represents a concrete value, its size is statically bounded by the kind of write being done. For example, a write to main memory requires 1 byte + 8 byte + 1 byte = 10 byte and modifying a 64 bit register requires 1 byte + 8 byte + $\frac{64\,\text{bit}}{8\,\text{bit/byte}}$ = 17 byte. In the case of fragments representing symbolic values on the other hand, such a guarantee cannot effectively be made, as the symbolic expression may become arbitrarily large. Consider, for example, a symbolic expression of the form $\lambda = \text{input}_1 + \text{input}_2 + \ldots + \text{input}_n$, whose result is directly influenced by an arbitrary amount of n input words.

In summary, fragments are created in a way that precludes structural weaknesses as long as the hash algorithm used (in our case 256 bit BLAKE2b) remains unbroken and collisions are significantly less probable than transient failures of the computer performing the analysis.

5.3 Fingerprint Tracking

When using the KLEE SymEx engine, the call stack is not explicitly mapped into the program's address space, but rather directly managed by KLEE itself. This enables us to further extend the practical usefulness of our analysis by only considering fragments that are directly addressable from each point of the execution, which in turn enables the detection of certain non-terminating recursive function calls. It also goes well together with the implicit cleanup of all function variables when a function returns to its caller.

To incrementally construct the current fingerprint we utilize a stack that follows the current call stack, as is shown exemplary in Fig. 3. Each entry consists of three different parts: 1. a (partial) fingerprint over all local registers, i.e., objects that are not globally addressable, 2. a (partial) fingerprint over all locally allocated objects in main memory and 3. a list of pairs of instruction IDs and fingerprints, that denote the states that were encountered previously.

Modifying Objects. Any instruction modifying an object without reading input, such as an addition, is dealt with as explained previously: First, recompute the hash of the old fragment(s) before the instruction is performed and remove it from the current fingerprint. Then, perform the instruction, compute the hash of the new fragment(s) and add it to the current fingerprint.

Similarly modify the appropriate partial fingerprint, e.g., for a `load` the fingerprint of all local registers of the current function. Note that this requires each memory object to be mappable to where it was allocated from.

Function Calls. To perform a function call, push a new entry onto the stack with the register fingerprint initialized to the xor of the hashes of the argument fragments and the main memory fingerprint set to the neutral element, zero. Update the current fingerprint by removing the caller's register fingerprint and adding the callee's register fingerprint. Add the pair of entry point and current fingerprint to the list of previously seen fingerprints.

Function Returns. When returning from a function, first remove both the fingerprint of the local registers, as well as the fingerprint of local, globally addressable objects from the current fingerprint, as all of these will be implicitly destroyed by the returning function. Then pop the topmost entry from the stack and re-enable the fingerprint of the local registers of the caller.

Reading Input. Upon reading input all previously encountered fingerprints must be disregarded by clearing all fingerprint lists of the current SymEx state.

5.4 Avoiding Comparisons

While it would be sufficient to simply check all previous fingerprints for a repetition every time the current fingerprint is modified, it would be rather inefficient to do so. To gain as much performance as possible, our implementation attempts to perform as few comparisons as possible.

We reduce the number of fingerprints that need to be considered at any point by exploiting the structure of the call stack: To find any non-recursive infinite loop, it suffices to search the list of the current stack frame, while recursive infinite loops can be identified using only the first fingerprint of each stack frame.

We also exploit static control flow information by only storing and testing fingerprints for Basic Blocks (BBs), which are sequences of instructions with linear control flow[6]. If any one instruction of a BB is executed infinitely often, all of them are. Thus, a BB is either fully in the infinite cycle, or no part of it is.

It is not even necessary to consider every single BB, as we are looking for a trace with a finite prefix leading into a cycle. As the abstract transition system is

[6] In IR there is an exemption for function calls, namely they do not break up BBs.

an unfolding of the CFG, any cycle in the transition system must unfold from a cycle in the CFG. Any reachable cycle in the CFG must contain a BB with more than one predecessor, as at least one BB must be reachable from both outside and inside the cycle. Therefore, it is sufficient to only check BBs with multiple predecessors. As IR only provides intraprocedural CFGs, we additionally perform a check for infinite recursion at the beginning of each function.

6 Evaluation

In this section we demonstrate the effectiveness and performance of our approach on well tested and widely used real-world software. We focus on three different groups of programs: 1. The GNU Coreutils and GNU `sed` (Sec. 6.1), 2. BusyBox (Sec. 6.2) and 3. toybox (Sec. 6.3) and evaluate the performance of our liveness analysis in comparison with baseline KLEE in the following metrics: 1. instructions per second and 2. peak resident set size. Additionally, we analyze the impact of the time limit on the overhead (Sec. 6.4). We summarize our findings in Sec. 6.5.
Setup. We used revision `aa01f83`[7] of our software, which is based on KLEE revision `37f554d`[8]. Both versions are invoked as suggested by the KLEE authors and maintainers [10,47] in order to maximize reproducability and ensure realistic results. However, we choose the Z3 [39] solver over STP [20] as the former provides a native timeout feature, enabling more reliable measurements. The solver timeout is 30 s and the memory limit is 10 000 MiB.

We run each configuration 20 times in order to gain statistical confidence in the results. From every single run, we extract both the instructions, allowing us to compute the instructions per second, and the peak resident set size of the process, i.e., the maximal amount of memory used. We additionally reproduced the detected liveness violations with 30 runs each with a time limit of 24 hr, recording the total time required for our implementation to find the first violation. For all results we give a 99 % confidence interval.

6.1 GNU Utilities

We combine the GNU tools from the Coreutils 8.25 [45] with GNU `sed` 4.4 [46], as the other tool suites also contain an implementation of the `sed` utility. We excluded 4 tools from the experiment as their execution is not captured by KLEE's system model. Thereby, the experiment contains a total of 103 tools.
Violations. The expected liveness violation in `yes` occurred after 2.51 s±0.26 s. In 26 out of 30 runs, we were also able to detect a violation in GNU `sed` after a mean computation time of 8.06 hr ± 3.21 hr (KLEE's timeout was set to 24 hr). With the symbolic arguments restricted to one argument of four symbolic characters, reproduction completed in all 30 runs with a mean of 5.19 min ± 0.17 min.

We detected multiple violations in `tail` stemming from two previously unknown bugs, that we reported. Both bugs were originally detected and reported in version

[7] `https://github.com/COMSYS/SymbolicLivenessAnalysis/tree/aa01f83`
[8] `https://github.com/klee/klee/tree/37f554d`

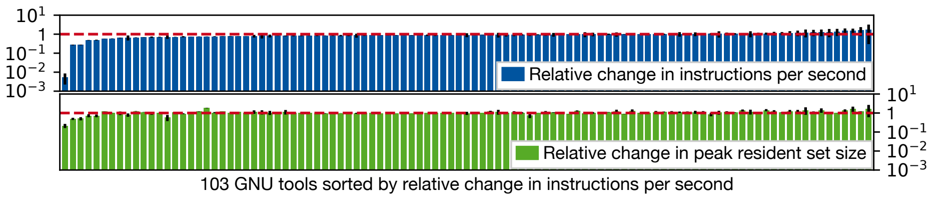


Fig. 4: GNU Coreutils and GNU `sed`, 60 min time limit. Relative change of instructions per second (top) and peak resident set (bottom) versus the KLEE baseline. Note the logarithmic scale and the black 99 % confidence intervals.

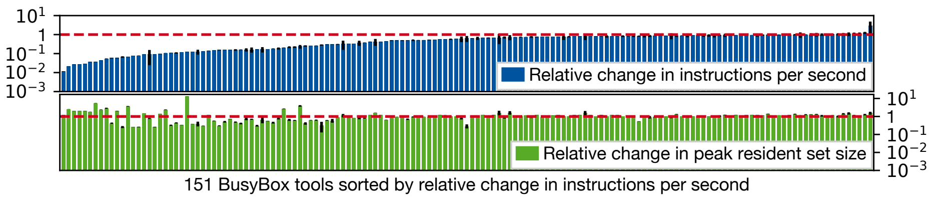


Fig. 5: BusyBox, 60 min time limit. Relative change of instructions per second (top) and peak resident set (bottom) versus the KLEE baseline. Note the logarithmic scale and the black 99 % confidence intervals.

8.25[9] and fixed in version 8.26. Both bugs were in the codebase for over 16 years. Reproducing the detection was successful in 30 of 30 attempts with a mean time of 1.59 hr $\pm$ 0.66 hr until the first detected violation.

We detected another previously unknown bug in `ptx`. Although we originally identified the bug in version 8.27, we reported it after the release of 8.28[10], leading to a fix in version 8.29. This bug is not easily detected: Only 9 of 30 runs completed within the time limit of 24 hr. For these, mean time to first detection was 17.15 hr $\pm$ 3.74 hr.

Performance. Fig. 4 shows the relative changes in instructions per second and peak resident set. As can be seen, performance is only reduced slightly below the KLEE baseline and the memory overhead is even less significant. The leftmost tool, `make-prime-list`, shows the by far most significant change from the KLEE baseline. This is because `make-prime-list` only reads very little input, followed by a very complex computation in the course of which no further input is read.

6.2 BusyBox

For this experiment we used BusyBox version 1.27.2 [44]. As BusyBox contains a large number of network tools and daemons, we had to exclude 232 tools from the evaluation, leaving us with 151 tools.

[9] GNU `tail` report 1: `http://bugs.gnu.org/24495`
GNU `tail` report 2: `http://bugs.gnu.org/24903`

[10] GNU `ptx` report: `http://bugs.gnu.org/28417`

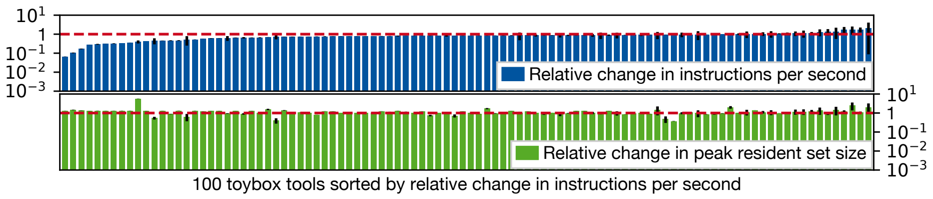


Fig. 6: Toybox, 60 min time limit. Relative change of instructions per second (top) and peak resident set (bottom) versus the KLEE baseline. Note the logarithmic scale and the black 99 % confidence intervals.

Violations. Compared with Coreutils' `yes`, detecting the expected liveness violation in the BusyBox implementation of `yes` took comparatively long with 27.68 s ± 0.33 s. We were unable to detect any violations in BusyBox `sed` without restricting the size of the symbolic arguments. When restricting them to one argument with four symbolic characters, we found the first violation in all 30 runs within 1.44 hr ± 0.08 hr. Our evaluation uncovered two previously unknown bugs in BusyBox `hush`[11]. We first detected both bugs in version 1.27.2. In all 30 runs, a violation was detected after 71.73 s ± 5.00 s.
Performance. As shown in Fig. 5, BusyBox has a higher slowdown on average than the GNU Coreutils (c.f. Fig. 4). Several tools show a *decrease* in memory consumption that we attribute to the drop in retired instructions. `yes` shows the least throughput, as baseline KLEE very efficiently evaluates the infinite loop.

6.3 Toybox

The third and final experiment with real-world software consists of 100 tools from toybox 0.7.5 [48]. We excluded 76 of the total of 176 tools, which rely on operating system features not reasonably modeled by KLEE.
Violations. For `yes` we encounter the first violation after 6.34 s ± 0.24 s, which puts it in between the times for GNU `yes` and BusyBox `yes`. This violation is also triggered from `env` by way of toybox's internal path lookup. As with the other `sed` implementations, toybox `sed` often fails to complete when run with the default parameter set. With only one symbolic argument of four symbolic characters, however, we encountered a violation in all 30 runs within 4.99 min ± 0.25 min.
Performance. Overall as well, our approach shows a performance for toybox in between those for the GNU Coreutils and BusyBox, as can be seen in Fig. 6. Both memory and velocity overhead are limited. For most toybox tools, the overhead is small enough to warrant always enabling our changes when running KLEE.

6.4 Scaling with the Time Limit

To ascertain whether the performance penalty incurred by our implementation scales with the KLEE time limit, we have repeated each experiment with time

[11] BusyBox `hush` report 1: `https://bugs.busybox.net/10421`
BusyBox `hush` report 2: `https://bugs.busybox.net/10686`

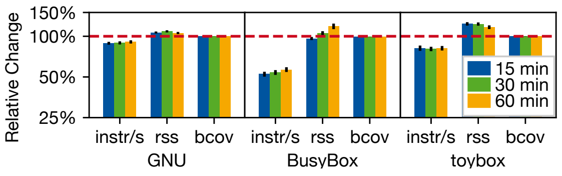


Fig. 7: Changes in instructions per second, peak resident set and branch coverage over multiple KLEE timeouts. Note the logarithmic scale and the black 99 % confidence intervals.

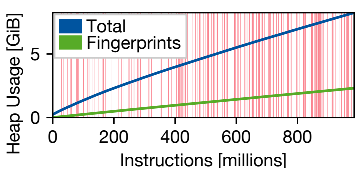


Fig. 8: Heap usage of a 30 min BusyBox `hush` run. The 186 vertical lines show detected liveness violations.

limits 15 min, 30 min and 60 min. The results shown in Fig. 7 indicate that, at least at this scale, baseline KLEE and our implementation scale equally well. This is true for almost all relevant metrics: retired instructions per second, peak resident set and covered branches. The prominent exception is BusyBox's memory usage, which is shown exemplary in Fig. 8 for a 30 min run of BusyBox `hush`. As can be seen, the overhead introduced by the liveness analysis is mostly stable at about a quarter of the total heap usage.

6.5 Summary

All evaluated tool suites show a low average performance and memory penalty when comparing our approach to baseline KLEE. While the slowdown is significant for some tools in each suite, it is consistent as long as time and memory limits are not chosen too tightly. In fact, for these kinds of programs, it is reasonable to accept a limited slowdown in exchange for opening up a whole new category of defects that can be detected. In direct comparison, performance varies in between suites, but remains reasonable in each case.

7 Limitations

Our approach does not distinguish between interpreters and interpreted programs. While this enables the automatic derivation of input programs for such interpreters as `sed`, it also makes it hard to recognize meaningful error cases. This causes the analysis of all three implementations of `sed` used in the evaluation (Sec. 6) to return liveness violations.

In its current form, our implementation struggles with runaway counters, as a 64 bit counter cannot be practically enumerated on current hardware. Combining static analyses, such as those done by optimizing compilers may significantly reduce the impact of this problem in the future.

A different pattern that may confound our implementation is related to repeated allocations. If memory is requested again after releasing it, the newly acquired memory may not be at the same position, which causes any pointers to it to have different values. While this is fully correct, it may cause the implementation to not recognize cycles in a reasonable time frame. This could be mitigated by analyzing whether the value of the pointer ever actually matters. For example, in the C programming language, it is fairly uncommon to inspect the numerical value of a pointer beyond comparing it to `NULL` or other pointers. A valid solution would however require strengthening KLEE's memory model, which currently does not model pointer inspection very well.

Another potential problem is how the PC is serialized when using symbolic expressions as the value of a fragment (c.f. Sec. 5.2). We currently reuse KLEE's serialization routines, which are not exactly tuned for performance. Also, each symbolic value that is generated by KLEE is assigned a unique name, that is then displayed by the serialization, which discounts potential equivalence.

Finally, by building upon SymEx, we inherit not only its strengths, but also its weaknesses, such as a certain predilection for state explosion and a reliance on repeated SMT solving [12]. Also, actual SymEx implementations are limited further than that. For example, KLEE returns a concrete pointer from allocation routines instead of a symbolic value representing all possible addresses.

8 Conclusion and Outlook

It is our strong belief that the testing and verification of liveness properties needs to become more attractive to developers of real-world programs. Our work provides a step in that direction with the formulation of a liveness property that is general and practically useful, thereby enabling even developers uncomfortable with interacting with formal testing and verification methods to at least check their software for liveness violation bugs.

We demonstrated the usefulness of our liveness property by implementing it as an extension to the Symbolic Execution engine KLEE, thereby enabling it to discover a class of software defects it could not previously detect, and analyzing several large and well-tested programs. Our implementation caused the discovery and eventual correction of a total of five previously unknown defects, three in the GNU Coreutils, arguably one of the most well-tested code bases in existence, and two in BusyBox. Each of these bugs had been in released software for over 7 years—four of them even for over 16 years—, which goes to show that this class of bugs has so far proven elusive. Our implementation did not cause a single false positive: all reported violations are indeed accompanied by concrete test cases that reproduce a violation of our liveness property.

The evaluation in Sec. 6 also showed that the performance impact, in matters of throughput as well as in matters of memory consumption, remains significantly below 2× on average, while allowing the analysis to detect a completely new range of software defects. We demonstrated that this overhead remains stable over a range of different analysis durations.

In future work, we will explore the opportunities for same-state merging that our approach enables by implementing efficient equality testing of SymEx states via our fingerprinting scheme. We expect that this will further improve the performance of our approach and maybe even exceed KLEE's baseline performance by reducing the amount of duplicate work done.

Acknowledgements

This research is supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation Programme (grant agreement №. 647295 (SYMBIOSYS)).

References

1. Alpern, B., Schneider, F.B.: Verifying temporal properties without temporal logic. ACM Trans. Program. Lang. Syst. 11(1), 147–167 (1989)
2. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21(4), 181–185 (1985)
3. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2(3), 117–126 (1987)
4. Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: Simpler, Smaller, Fast as MD5. In: Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS'13). pp. 119–135 (Jun 2013)
5. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
6. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. International Journal on Software Tools for Technology Transfer 5(1), 49–58 (2003)
7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Springer (1999)
8. Borralleras, C., Brockschmidt, M., Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination through conditional termination. In: TACAS 2017. pp. 99–117. Springer (2017)
9. Burnim, J., Jalbert, N., Stergiou, C., Sen, K.: Looper: Lightweight Detection of Infinite Loops at Runtime. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09) (Nov 2009)
10. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08) (Dec 2008)
11. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically Generating Inputs of Death. ACM Transactions on Information and System Security 12(2), 10 (2008)
12. Cadar, C., Sen, K.: Symbolic Execution for Software Testing: Three Decades Later. Communications of the ACM 56(2), 82–90 (2013)
13. Chan, W., Anderson, R.J., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J.D.: Model checking large software specifications. IEEE Transactions on Software Engineering 24(7), 498–520 (1998)
14. Chen, H.Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.W.: Proving Nontermination via Safety. In: Proceedings of the 20st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14) (Apr 2014)

15. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11) (Mar 2011)
16. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 2004. pp. 168–176. Springer (2004)
17. Clarke, E.M., Grumberg, O., Hiraishi, H., Jha, S., Long, D.E., McMillan, K.L., Ness, L.A.: Verification of the Futurebus+ cache coherence protocol. Formal Methods in System Design 6(2), 217–232 (1995)
18. Cordeiro, L., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ansi-c software. IEEE Transactions on Software Engineering 38(4), 957–974 (2012)
19. Falke, S., Merz, F., Sinz, C.: The bounded model checker LLBMC. In: ASE 2013. pp. 706–709 (2013)
20. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Proceedings of the 19th International Conference on Computer-Aided Verification (CAV'07). pp. 519–531 (Jul 2007)
21. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL '97. pp. 174–186. ACM (1997)
22. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05). vol. 40, pp. 213–223 (Jun 2005)
23. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox Fuzzing for Security Testing. ACM Queue 10(1), 20 (2012)
24. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated Whitebox Fuzz Testing. In: Proceedings of the 15th Network and Distributed System Security Symposium (NDSS'08). vol. 8, pp. 151–166 (Feb 2008)
25. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'08) (Jun 2008)
26. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL '08. pp. 147–158. ACM (2008)
27. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer 2(4), 366–381 (2000)
28. Hensel, J., Giesl, J., Frohn, F., Ströder, T.: Proving termination of programs with bitvector arithmetic by symbolic execution. In: SEFM 2016. pp. 234–252. Springer (2016)
29. Holzmann, G., Najm, E., Serhrouchni, A.: Spin model checking: an introduction. International Journal on Software Tools for Technology Transfer 2(4), 321–327 (2000)
30. Holzmann, G.J.: Design and validation of protocols: a tutorial. Computer Networks and ISDN Systems 25(9), 981–1017 (1993)
31. Kindler, E.: Safety and liveness properties: A survey. Bulletin of the European Association for Theoretical Computer Science 53, 268–272 (1994)
32. Kling, M., Misailovic, S., Carbin, M., Rinard, M.: Bolt: On-demand Infinite Loop Escape in Unmodified Binaries. In: Proceedings of the 27th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'12) (Oct 2012)

33. Kroening, D., Ouaknine, J., Strichman, O., Wahl, T., Worrell, J.: Linear completeness thresholds for bounded model checking. In: CAV 2011. pp. 557–572. Springer (2011)
34. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design 19(3), 291–314 (2001)
35. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving Non-termination Using Max-SMT. In: Proceedings of the 26th International Conference on Computer-Aided Verification (CAV'14) (Jul 2014)
36. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO'04). pp. 75–88. San Jose, CA, USA (Mar 2004)
37. Liew, D., Schemmel, D., Cadar, C., Donaldson, A.F., Zähl, R., Wehrle, K.: Floating-Point Symbolic Execution: A Case Study in N-Version Programming. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17). pp. 601–612 (Oct-Nov 2017)
38. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: Verified Software: Theories, Tools, Experiments. pp. 146–161. Springer (2012)
39. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08). pp. 337–340 (Mar-Apr 2008)
40. Owicki, S., Lamport, L.: Proving liveness properties of concurrent programs. TOPLAS 4(3), 455–495 (1982)
41. Sasnauskas, R., Landsiedel, O., Alizai, M.H., Weise, C., Kowalewski, S., Wehrle, K.: KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'10) (Apr 2010)
42. Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05). vol. 30, pp. 263–272 (Jun 2005)
43. Straunstrup, J., Andersen, H.R., Hulgaard, H., Lind-Nielsen, J., Behrmann, G., Kristoffersen, K., Skou, A., Leerberg, H., Theilgaard, N.B.: Practical verification of embedded software. Computer 33(5), 68–75 (2000)
44. The BusyBox Developers: BusyBox: The Swiss Army Knife of Embedded Linux. https://busybox.net (Aug 2017), version 1.27.2
45. The GNU Project: GNU Coreutils. https://www.gnu.org/software/coreutils (Jan 2016), version 8.25
46. The GNU Project: GNU sed. https://www.gnu.org/software/sed (Feb 2017), version 4.4
47. The KLEE Team: OSDI'08 Coreutils Experiments. http://klee.github.io/docs/coreutils-experiments/, Section 07
48. The toybox Developers: toybox. http://landley.net/toybox (Oct 2017), version 0.7.5
49. Tillmann, N., De Halleux, J.: Pex–White Box Test Generation for .NET. In: Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08). pp. 134–153 (Apr 2008)
50. Tretmans, J., Wijbrans, K., Chaudron, M.: Software engineering with formal methods: The development of a storm surge barrier control system revisiting seven myths of formal methods. Formal Methods in System Design 19(2), 195–215 (2001)