Triereme: Speeding up hybrid fuzzing through efficient query scheduling

Elia Geretto[*]
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
e.geretto@vu.nl

Julius Hohnerlein[*]
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
julius.hohnerlein@posteo.de

Cristiano Giuffrida
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
herbertb@cs.vu.nl

Erik van der Kouwe
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
vdkouwe@cs.vu.nl

Klaus v. Gleissenthall
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
k.freiherrvongleissenthal@vu.nl

ABSTRACT

Hybrid fuzzing, the combination between fuzzing and concolic execution, holds great promise in theory, but has so far failed to deliver all the expected advantages in practice due to its high overhead. The cause is the large amount of time spent in the SMT solver. As a result, hybrid fuzzers often lose out to simpler, yet faster techniques. This issue remains despite novel query pruning techniques that reduce the number and complexity of solver queries as they preclude other crucial optimizations like incremental solving.

We introduce Triereme, a method to speed up the hybrid fuzzer's concolic engine by reducing the time spent in the SMT solver. Triereme uses a trie (or prefix tree) data structure to schedule and cache solver queries, exploiting common prefixes. This design is made possible by decoupling concolic *tracing* from concolic *solving*. As a result, Triereme manages to reconcile pruning with incremental solving, reaping their combined benefits. In our tests, Triereme speeds up concolic executions by 6.1x on average in FuzzBench [22] and improves coverage progress in 79% of the benchmarks.

CCS CONCEPTS

• Security and privacy → Software security engineering; **• Software and its engineering → Software testing and debugging**.

KEYWORDS

fuzzing, hybrid fuzzing, concolic execution, program analysis

ACM Reference Format:
Elia Geretto, Julius Hohnerlein, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus v. Gleissenthall. 2023. Triereme: Speeding up hybrid fuzzing through efficient query scheduling. In *Annual Computer Security Applications Conference (ACSAC '23), December 4–8, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627106.3627173

[*]Both authors contributed equally to the paper.

1 INTRODUCTION

Despite the intuitive advantages, few fuzzing projects outside of academic research adopt hybrid fuzzing, the combination between fuzzing and concolic execution [27]. In theory, this combination is attractive: fuzzers can quickly explore code with simple branches that are likely to flip with random inputs, while concolic engines can use SMT solvers to solve the more complex cases, too hard for traditional fuzzers. In practice, the benefits often fail to materialize and hybrid fuzzers are routinely outperformed by conceptually simpler, yet faster techniques [1]. The problem is that the overhead of concolic execution cancels out any potential gains. To realize hybrid fuzzing's promise, we need to speed it up [8, 23, 24, 32].

After removing important bottlenecks in the construction and management of symbolic expressions [8, 23, 24], the bulk of the overhead is now due to the time spent in the SMT solver [32]. Indeed, our experiments show that in our baseline configuration, solving takes up 59% of concolic execution on average. To reduce the impact of the solver, the community proposed various pruning techniques to simplify and reduce the number of SMT solver queries [7, 9, 32]. Yet, the overhead remains. Indeed, despite being overall beneficial, pruning and simplification preclude the use of incremental solving, which is a key ingredient for solver performance [28].

In this paper, we propose Triereme, a concolic execution engine that reduces the time spent running the SMT solver by introducing a new way to organize solver queries based on a trie (or prefix tree) data structure [4]. This reorganization, made possible by *decoupling* tracing and solving in separate processes, allows Triereme to make optimal use of the solver's incremental solving capabilities, even when combined with modern pruning and simplification methods [7, 9, 32]. Finally, the trie also caches query results between different concolic executions, enabling further optimizations.

In our implementation, Triereme reduces the time spent using the SMT solver by 63% on average in FuzzBench [22] and achieves a speedup of up to 15.2x for the complete concolic execution over our SymCC baseline. As a result, Triereme improves coverage or the speed at which we reach coverage in 79% of the benchmarks.

In summary, we make the following contributions:

- We present Triereme, a new trie-based approach to solve the query scheduling problem for concolic execution engines.
- We implement a prototype of Triereme, which is available at https://github.com/vusec/triereme.

```
bool is_sorted(size_t *a, size_t n) {
  for (size_t i = 0; i + 1 < n; i++) {
    if (a[i] > a[i + 1]) {
      return false;
    }
  }
  return true;
}
```

Listing 1: A function that checks whether an array is sorted.

```
1 >= n
1 < n && a[0] > a[1]
1 < n && a[0] <= a[1] && 2 >= n
1 < n && a[0] <= a[1] && 2 < n && a[1] > a[2]
1 < n && a[0] <= a[1] && 2 < n && a[1] <= a[2]
  && 3 < n
```

Listing 2: Query schedule generated by a concolic execution of Listing 1 with the array `[1,2,3]`.

- We evaluate the effectiveness of our approach, comparing it with an improved SymCC baseline and AFL++, a state-of-the-art fuzzer. It achieves up to 15.2x speedup over our baseline, with better coverage progress in 79% of the benchmarks.

2 BACKGROUND: CONCOLIC EXECUTION

Concolic execution is a dynamic program analysis technique that instruments a concrete execution to create symbolic expressions for each variable in the program. Collectively, these expressions form the symbolic state.

As in symbolic execution, the concolic engine collects constraints over the symbolic state at every branch. These constraints describe the current control-flow path through the program. Unlike symbolic execution, though, a concolic engine does not fork on branches to explore both alternatives, but instead only follows one path—that of the concrete execution. To explore the alternative branch, it produces a new test case that satisfies the path constraint for the *alternative* outcome of the branch, i.e., by conjoining the past constraints leading up to the branch with the negation of the current branch constraint. When executing this new test case, the concrete execution will follow the alternative path, increasing coverage [15].

Importantly, existing instrumentations solve the constraints by querying an SMT solver such as Z3 [11] *inline*, that is, as soon as a new branch is encountered. This approach produces a query schedule that is dependent on the execution flow.

As an example, consider Listing 1 and a concrete execution that calls `is_sorted()` with the array `[1,2,3]`, leading to the path constraints in Listing 2. The constraints progress along the pattern $\neg A$, $A \land \neg B$, $A \land B \land \neg C$, etc. Apart from the last branch constraint, the schedule has a common prefix that grows with every branch.

Incremental solving. The repetition of the prefix for a single execution has allowed the authors of earlier concolic execution engines such as Driller [27] and S2E [10] to reduce the amount of time spent solving through *incremental solving*, which allows the SMT solver to reuse lemmas learned from previous similar queries [32]. In particular, SMT solvers are capable of pushing and

```
1 >= n
a[0] > a[1]
1 < n && 2 >= n
a[0] <= a[1] && a[1] > a[2]
1 < n && 2 < n && 3 < n
```

Listing 3: Query schedule generated by using the unrelated constraint elimination technique on the query schedule in Listing 2.

popping specific states on an assertion stack. The assertion stack makes it easy to go from, say, $A \land \neg B$ to $A \land B \land \neg C$ by (1) pushing A, (2) pushing $\neg B$, (3) popping $\neg B$, and pushing $B \land \neg C$, which can then be solved using the knowledge obtained earlier from A.

Any changes to the common prefix require resetting the solver state, leading to the loss of all knowledge acquired up to that point. Thus, incremental solving highly depends on the order in which the path constraints are solved.

Constraint filtering. Limiting the number and complexity of queries is important to limit the amount of time spent running the SMT solver. For instance, other works [24, 25, 32] use a coverage map to eliminate redundant queries across executions and limit the complexity of queries by removing unrelated constraints (i.e., constraints that transitively do not share any variables with the current path constraint we want to flip). Although this technique may result in the production of some useless test cases, it ensures performance gains that guarantee better coverage overall [32].

Applying this technique for complexity reduction to the queries in Listing 2 would produce the query schedule in Listing 3. While the reduction in query complexity produces overall faster solving times than the naive solution, the speedups are not significant enough to eliminate SMT solving as the bottleneck. Additionally, the elimination of the constraints also destroys the shared prefix which stops us from reaping the benefits of incremental solving.

3 OVERVIEW

In this paper, we propose Triereme, a concolic engine that reduces the time spent running the SMT solver based on two main steps. First, like SAGE [16], Triereme *decouples* the construction of solver queries (the *tracing* phase) from their execution (the *solving* phase). This way, query schedules are not bound by the execution flow, as with inline concolic execution engines. This allows us to transfer, reorder, and store solver queries and their results within and across executions. For example, Triereme can avoid solving a query if the same query was already issued previously or if the query is an extension of a query that was previously judged unsatisfiable.

Second, Triereme organizes and solves queries using a *trie* (or *prefix tree*) data structure which allows it to make optimal use of incremental solving. Our trie traversal algorithm manipulates the assertion stack at each trie node so that the solver can always exploit common prefixes. As a result, Triereme can use path constraint filtering techniques implemented in many symbolic and concolic execution engines [7, 9, 32], such as the elimination of unrelated constraints, without suffering from their main performance drawback: the inability to use incremental solving.

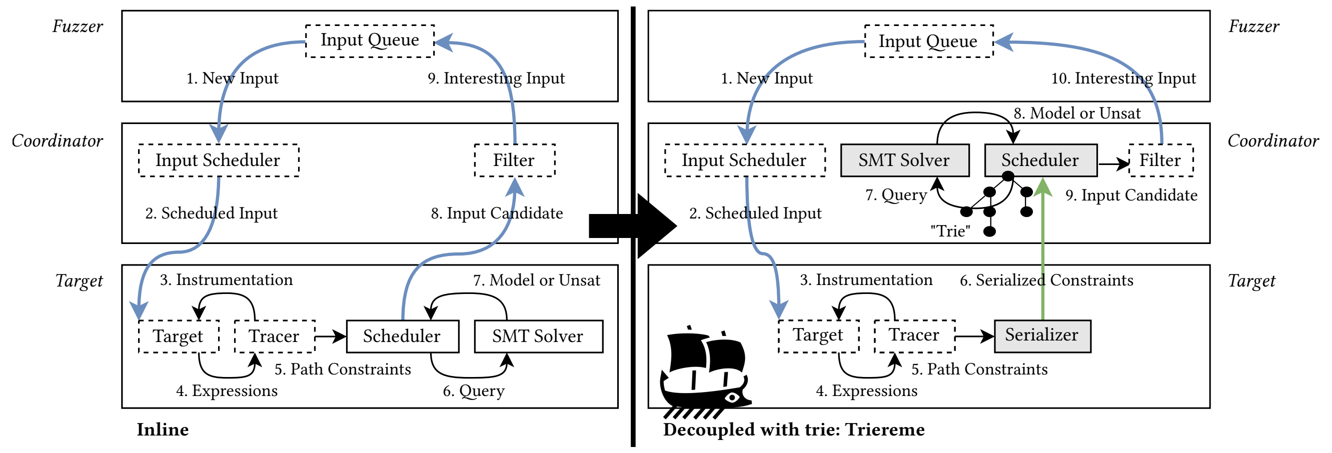


Figure 1: Conceptual information flow in *inline* (left) vs. *decoupled/TRIEREME* design (right). By moving the scheduler and the SMT solver into the Coordinator, TRIEREME optimizes the schedule while preserving incremental solving. Small boxes represent components, and large ones processes. Arrows represent information flow, where black, blue, and green indicate where we use heap memory, the file system, and shared memory as the information medium respectively. Small dashed boxes are unchanged.

Figure 1 shows the main steps performed in our engine, including the changes that decouple the construction of solver queries from their execution. The input scheduling portion of the execution loop is the same as in the inline design. A test case is taken from the queue and fed to the instrumented program under test. The main difference starts in the instrumentation itself. In the inline design, the instrumentation constructs symbolic expressions, produces solver queries, and uses the SMT solver to find solutions, in lock-step with the concrete execution. The output of the instrumentation is a series of candidate test cases that will be filtered. Instead, in the decoupled design of TRIEREME, the concolic execution gathers symbolic expressions for the branch constraints and streams them in serialized form to the coordinator through shared memory. Once the program under test finishes executing, the scheduler deserializes the expressions in its own address space and processes the resulting branch constraints according to its path simplification strategy; it then organizes branch constraints into a trie data structure that it uses to schedule constraint queries to the SMT solver. The engine produces candidate test cases only at this point. Finally, TRIEREME filters the candidates to verify that they indeed produce new coverage, as in the inline design.

4 DESIGN

This section discusses TRIEREME's constraint processing pipeline, the trie data structure used to store path constraints, and the optimizations enabled by the trie. Unlike existing work, our pipeline processes new path constraints centrally, allowing us to reason over the entire set of constraints and schedule solving more efficiently.

4.1 Constraint Processing Pipeline

While running the target program, TRIEREME gathers path constraints to later target conditions that can flip branches encountered along the way. TRIEREME's constraint processing pipeline extracts branch constraints from the target program, filters, and assembles them. Afterwards, we insert these constraints into our trie data structure for solving and caching (see 4.2).

Tracing. TRIEREME instruments the target program by inserting calls to its runtime library, which collects expressions based on the symbolic values of variables. While we are only interested in branch constraints, TRIEREME instruments all instructions, as any operation might affect a later branch constraint. Branch constraints are specifically marked as such. From here, the expressions are passed to the serialization part of TRIEREME.

TRIEREME concretizes expressions that likely belong to loops during tracing. It identifies these expressions by keeping a hitcount map for basic blocks, so that it can detect basic blocks that have been traversed multiple times. This step has proven effective in QSYM, and in the case of TRIEREME it helps limit the amount of data transferred between the tracer and scheduler.

Serialization. In an inline design, path constraints are immediately solved in the process where they were collected. To decouple tracing and solving, TRIEREME instead serializes the expressions to pass them to its scheduling component. We designed a novel serialization format that is simple, space-efficient, and minimizes serialization and transmission overhead. The format is also resilient to the target abruptly ending execution, which is expected in fuzzing.

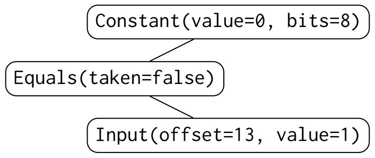
Figure 2 shows an example of the encoding of a concolic trace using this format. Expressions are serialized as they are encountered, in program execution order; this guarantees that the sub-expressions that are used to construct a new expression are always already present in the trace. We retrieve the expressions referenced using SymCC, which uses local variables to track expressions in local variables, and shadow memory to track expressions in heap memory [24]. We refer to sub-expressions of a new expression using an offset that provides the relative position of the sub-expression in the serialized expression stream. Since sub-expressions typically refer to recent nodes in actual executions, the offset is usually small.

```c
if (inp[13] == 0) {
    // ...
}
```

(a) A snippet of C code,

```
[0]: Constant(value=0, bits=8)
[1]: Input(offset=13, value=1)
[2]: Equals(l=[-2], r=[-1], taken=false)
```

(b) ... its serialized form,



(c) ... its AST representation during a concolic execution.

Figure 2: The serialization format by example. The implicit running identifiers are shown at the beginning of each line in Figure 2b, next to the expression that they identify.

In combination with variable-length integer encoding, the small offset means that the tracer can encode a typical binary operation in just 3 bytes (1 byte for the operation and 1 byte each for the operands). The last line in Figure 2b demonstrates the use of offset references. This expression references the previous two expressions, offset -2 on the left-hand side and offset -1 on the right-hand side. During our development, we observed trace sizes of about 1 MB on average and 5 MB maximum.

Path constraint construction. The instrumentation sends serialized constraints to our scheduler process, which deserializes the constraints into Abstract Syntax Trees (ASTs), one per expression (see Figure 2c). This encoding is loosely based on LLVM IR.

To save memory, we fold constants in expressions when possible, similarly to [2]. In addition, their deserialization does not happen in isolation: when processing a new expression, the engine has already deserialized all of its sub-expressions. We reuse existing AST nodes where possible, effectively deduplicating sub-expressions.

At this point, the engine assembles branch constraints into path constraints for unexplored branches. This assembly is done by appending the appropriate negated branch constraints in order to obtain $\neg A$, $A \wedge \neg B$, $A \wedge B \wedge \neg C$, etc., as described in Section 2. The rest of the pipeline conceptually operates on these path constraints instead of expressions that make up each branch constraint.

Branch pruning. Not all path constraints are worth exploring. We avoid targeting branches that a test case has already covered. To achieve this, Triereme maintains a map of the total coverage across all concolic executions. We only keep branch constraints of explored branches to serve as a pre-condition for nested branches.

Path simplification. If a branch was not discarded, Triereme simplifies its associated path constraint according to the unrelated constraint filtering (see Section 2). Finally, Triereme inserts the simplified path constraints associated with interesting branches into the solver trie, and starts the solving phase. This pipeline is guaranteed to run at least every 90 s, the same as QSYM's execution timeout, even if the trie still has path constraints to be solved. This limit ensures a continuous supply of new path constraints which refer to new test cases; solving these new path constraints is more likely to help the fuzzer paired with the concolic engine as they are more likely to refer to conditions that the fuzzer has not been autonomously able to solve yet.

Discussion. Our decoupled design trades a small additional overhead, produced by the storage and transmission of constraints between processes, against a significant increase in flexibility. The traditional inline design uses very short-lived constraints that have to be processed as they are encountered. Although this reduces resource usage when tracing, it forces a specific query schedule and does not allow constraints or results to survive between executions. Our decoupled design, instead, uses long-lived constraints, which are more expensive to keep in memory, but can be easily reorganized and cached, as they survive individual concolic executions. Paying the price of additional flexibility is convenient only when this flexibility enables a performance benefit that covers the overhead, as we will show is the case for Triereme.

Keeping constraints in memory may seem infeasible due to the large number of expressions generated by a single execution. However, prior work [32] shows that aggressive expression pruning is necessary to make solving practical even in the more space-efficient inline design. Triereme shows that its pruning and deduplication can keep memory pressure low enough to be practical, even when combining constraints across executions. In our experiments, we have observed a maximum memory usage of 8.7 GB (see Table 7).

4.2 Trie-Based Solver Scheduling

Triereme exploits its decoupled design by introducing a scheduler based on the trie data structure. The trie organizes path constraints in a tree that maintains the following invariant: for any node in the tree, its children do not share a common prefix. This means that for any two path constraints with a common prefix, the prefix must correspond to a unique path through the graph. Scheduling queries using a depth-first search traversal thus guarantees optimal use of the single assertion stack available for incremental solving. The trie also helps in limiting memory usage as it shares internal nodes among path constraints that share prefixes.

Figure 3 shows a trie representation of the query schedules in Listing 3. Path constraints are stored in the trie as paths from the root to a node in bold; we call these bold nodes *path terminators.* The trie will contain one path terminator node per branch we are trying to solve, representing the negated branch constraint. All leaves must be path terminators, while interior nodes may or may not be path terminators. An interior node can become a path terminator if there is a path constraint that is a prefix of another path constraint.

Using the trie, Triereme schedules path constraints using Algorithm 1. The algorithm performs a depth-first traversal of the trie, restoring the solver state of each intermediate node before exploring any of its sub-tries. We achieve this by pushing the solver state onto the assertion stack before exploring a sub-trie and popping it immediately afterward. The engine produces a new candidate test case for each satisfiable constraint. We measure the resulting reduction of the length of path constraints in Section 6.

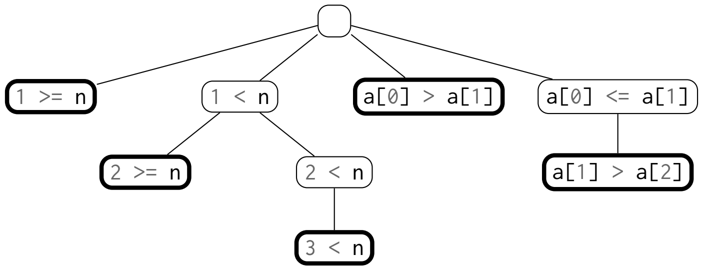


Figure 3: Constraint trie for query schedule in Listing 3. The nodes along all paths from the root to a node with a thick border represent a solver query, with each node along the path being a pre-condition to its successor.

Algorithm 1 Schedules the solver given a trie of constraints. Calls are modelled after the Z3 API, thus *PUSH* and *POP* refer to the assertion stack.

```
function SOLVE(node, results={})
    if node.is_path_terminator then
        results ← results ∪ { CHECK( ) }
    end if
    for child ∈ trie_node.children do
        PUSH( )
        ASSERT(node.constraint)
        SOLVE(child, results)
        POP( )
    end for
end function
```

4.3 Trie-Based Optimizations

Apart from incremental solving, the trie data structure enables other advanced caching and scheduling optimizations that are not viable in a traditional *inline* concolic engine.

Satisfiability result caching. For each path constraint, we store whether it is either *unsolved*, *solved and unsatisfiable*, or *solved and satisfiable*, along with the solution (where applicable). This information allows us to recall a solution efficiently if a duplicate satisfiable query is inserted or to detect early that a query is unsatisfiable.

Duplicate queries are possible despite the branch pruning step discussed in Section 4.1 because the pruning happens before path simplification, which eliminates portions of the path constraints. As a consequence, branches that have different addresses in the target program may result in the same simplified path constraint.

Unsatisfiability derivation. Path constraints assume the form $A \wedge B \wedge \ldots$, where $A, B, \ldots$ are branch constraints. When a path constraint is an extension of another constraint that has been proven unsatisfiable, we can conclude that the extension will also be unsatisfiable, as the $\wedge$ operator can only restrict the solution space. Reconstructing path constraints during the traversal of a trie allows us to easily identify these situations: when a node is associated with an unsatisfiable path constraint, all of its children will automatically be marked as unsatisfiable, without having to query the SMT solver. This optimization also works for path constraints added in future concolic executions, since we cache SMT results in the trie nodes.

Greedy smallest-subtrie-first scheduling. The order in which children are visited in the trie determines the order of solved constraints and, therefore, the solving schedule. We rely on the simple heuristic that in general, shorter paths are easier to solve. Therefore, we visit sub-tries in ascending size order. Here, the size of a sub-trie is the total number of elements, i.e., constraints.

While this scheduling solution is not important when all queries generated by a concolic execution finish within the solving period of 90 s mentioned in Section 4.1, it is important for programs in which this is not the case. When the trie scheduler is under pressure because it receives more path constraints than the SMT solver can process, smallest-subtrie-first scheduling will guarantee that fresh and fast path constraints will be prioritized. The trie scheduler will process longer constraints, which are more likely to timeout or prove unsatisfiable when it is no longer under pressure. In the context of hybrid fuzzing, this approach reduces the risk that the concolic engine produces test cases that have already been superseded by test cases from the fuzzer.

Optimistic pruning. Optimistic pruning continually removes the nodes in the trie using a "decay" mechanism to reduce memory pressure. Upon insertion into the trie, the mechanism assigns a generation number to each constraint and increments the number with each processed test case. When the solver follows a path in the trie, the mechanism resets the generation number of all constraints on that path. A configurable maximum generation number determines when the mechanism removes older nodes from the trie. This decay mechanism ensures that necessary nodes for solving survive across executions while huge tries are pruned effectively.

If a node is scheduled for removal when the SMT solver has not yet processed its path constraint, we do not simply erase it, but instead schedule it for optimistic solving, which we describe below. This gives TRIEREME another chance to solve the constraint, while optimistic solving ensures a simple query to the SMT solver.

Dynamic timeouts. In concolic execution engines that use *inline* solving, it is common to set a fixed timeout for each solver query. While this limit can be manually adjusted, it is difficult to collect statistics about the correct value for the program under test, as the solver runs together with the target program. With our decoupled design, it is possible to easily collect statistics about common solving times and adjust the timeout dynamically, similar to how fuzzers such as AFL [33] or AFL++ [12] adjust their hang detection timeout. After a warm-up period of 1000 concolic executions, we set the timeout to twice the value of the 95th percentile. During development, we observed a reduction of the default timeout period from the common default of 10 s to values between 1 s and 5 ms. Despite the large reduction of the threshold, the number of timeouts rose at most to only 4.9% of all queries with our Trie configuration, compared to the maximum of 1.2% obtained with our Linear configuration (see Table 9). This dynamic limiting is very useful in programs that generate many timeouts since even a small number of timeouts is sufficient to put the concolic execution engine under pressure. We also considered collecting statistics about query times on a subset of our suite and then set a single optimized timeout threshold for all benchmarks. However, we discarded this solution as the optimal timeout thresholds vary too much across benchmarks; a dynamic approach is thus preferable.

Optimistic solving. In addition to solving complete path constraints, we also implement optimistic solving [32]. In this case, we try to solve only the last branch constraint for an unsatisfiable path constraint. For example, if the path constraint $A \wedge \neg B$ is unsatisfiable, optimistic solving would try only with $\neg B$. This simplification is a way to get around some of the environment modeling limits present in concolic execution. We use a separate solver instance for optimistic solving to preserve the state of the assertion stack generated by the exploration of the trie.

5 IMPLEMENTATION

Triereme. Our Triereme prototype is based on SymCC [24] as we reuse its instrumentation code. Therefore, Triereme is compatible with the same benchmarks as SymCC and it shares SymCC's limitations, such as incomplete symbolic expressions with inline assembly. If SymCC receives compatibility improvements, Triereme can be easily adapted to benefit from them.

Most of our implementation effort is in the runtime library and the coordination component. Our runtime library is significantly smaller than the original QSYM-based runtime library provided by SymCC, since it only needs to record, serialize, and transmit the symbolic expressions it encounters. We added most of the code to the coordination component, which, in Triereme, performs most of the tasks originally performed by the instrumented program. This component shares some features with QSYM [32], but we implemented it from scratch. We implemented Triereme mainly in Rust, using around 6'000 lines of code.

Triereme always follows the pipeline illustrated in Section 4.1 to collect and assemble constraints. However, it can operate in two different modes as far as the query scheduler is concerned: the queries can be either processed using the trie scheduler described in Section 4.2 or using a "linear" scheduler. The linear scheduler replicates the behavior of a concolic execution engine that uses *inline* solving, like QSYM. This linear solving mode allows us to separate the performance overhead introduced by decoupling tracing and solving from the performance benefits derived from improved scheduling. We name these two modes *trie solver* and *linear solver.*

Shared changes. Apart from implementing Triereme, we also introduced modifications to the SymCC version we used as a baseline in our evaluation to bring its performance on par with the state of the art. These changes are replicated identically in Triereme.

First, we modified SymCC to work correctly with AFL++ [12], instead of the original, and now reasonably outdated, AFL [33]. As a consequence, we also reimplemented the filtering of candidate test cases using LibAFL [13], removing the use of `afl-showmap` from the coordinator component in SymCC. This change was necessary due to performance issues in `afl-showmap`, which would have significantly influenced the results of our experiments.

Second, we reduced the minimum synchronization interval in AFL++ from the default 30 minutes to 5 minutes. The AFL++ authors selected the default value to fit scenarios where tens of AFL++ instances share test cases. We found the new value more suited for our evaluation since it makes the fuzzer more sensitive to progress produced by the concolic engine. We observed an overall coverage increase due to this change during development.

6 EVALUATION

We run our evaluation using FuzzBench [22], which provides a set of predefined harnesses for real-world libraries. Since researchers used these harnesses for thousands of hours, it is almost impossible to find new bugs in a single campaign. From the 24 coverage benchmarks available, we extracted 14 benchmarks that are marked as supported by SymCC. We were forced to restrict the suite to what is compatible with our baseline, but the number of benchmarks is still in line with recommendations in the literature [5].

Our evaluation was run mostly on the cloud infrastructure provided by the FuzzBench project using `n1-standard-1` VMs with one vCPU core and 3.75 GB of RAM, which is FuzzBench's default. We evaluated `libjpeg`, `mbedtls`, and `vorbis` in a local FuzzBench experiment due to the limited amount of RAM available in the cloud VMs. The local machines have an AMD Ryzen Threadripper 2990WX and 128 GB of RAM. The local experiments also received one core, pinned to each trial. All trials lasted 23 hours, similarly to other works relying on the FuzzBench cloud infrastructure [5]; we repeated cloud experiments 20 times and local ones 16 times, for better core allocation. In both cases, we adhered to the best practices proposed by the literature [5, 18], which require a sufficient number of runs to verify statistical significance (see Table 10).

In our first experiment, we used only one core per trial because this setup is more representative of real-world campaigns that strive towards full utilization of all available cores. In particular, we used a hybrid fuzzing setup with a single AFL++ instance and a single instance of the concolic engine which share that single core. In this scenario, the fuzzer and the concolic engine compete for resources, so a concolic engine that is more performant both contributes more coverage progress and leaves more resources for the fuzzer when the engine remains idle. As this setup is different from the previous literature [24, 32], which dedicates a separate core for each component of the hybrid fuzzer, we further explored the implications of our setup in a second experiment.

We run AFL++ in fork mode with the common `PCGUARD` and `CMPLOG` options; the latter is a reimplementation of the technique proposed by RedQueen [1]. Although FuzzBench also supports persistent mode, our SymCC baseline does not support it and, consequently, neither does Triereme. Persistent mode would have created a significant imbalance in the efficiency of the two components of the hybrid system. As a result, the progress obtained by the fuzzer would have obscured improvements in the concolic execution engine. In addition, fork mode is most commonly used in the hybrid fuzzing literature [8, 24, 32].

As the purpose of TRIEREME is to solve roughly the same queries as our baseline, but faster, we have decided not to investigate its improvements in bug finding capabilities. Since we are not introducing new bug finding techniques, these improvements are the result of our increased exploration speed, which can be measured directly and more reliably with coverage [14, 18].

We run our experiments with four different configurations:

AFL++ This configuration includes only AFL++, i.e., just the fuzzer without concolic execution engine.

Baseline This configuration is our baseline concolic engine. It includes AFL++ and SymCC, run with its QSYM backend. To ensure a fair comparison, we applied performance improvements unrelated to TRIEREME (see Section 5).

Linear This configuration is a reimplementation of QSYM based on our codebase. It includes AFL++ and TRIEREME, with its linear backend. This backend relies on TRIEREME's constraint processing pipeline (see Section 4.1), but schedules constraints as an *inline* solver would do. Importantly, this configuration does not use the trie data-structure.

Trie This configuration is our proposed method. It includes AFL++ and TRIEREME, with our decoupled design and the trie scheduler described in Section 4.2, with all optimizations.

We do not include solutions using incremental solving that do not implement SymCC's optimizations [10, 27], as we cannot tell if the performance improvements we observe are due to our trie-based solution or SymCC's optimizations, which TRIEREME integrates.

In this evaluation, we answer the following three questions:

Q1 Does the solver trie significantly reduce the length and the number of queries to the SMT solver due to incremental solving and optimizations that operate across executions?

Q2 Does the solver trie reduce the amount of time spent running the SMT solver and, as a consequence, the amount of time necessary to process a single test case, saving computational resources that can be used by the fuzzer?

Q3 Do our optimizations improve coverage progress compared to SymCC with its QSYM backend?

Q4 To what extent will an alternative configuration, with a dedicated core to both the fuzzer and the concolic engine, impact CPU utilization (e.g., due to idling) and coverage?

6.1 Q1: Solver Queries

TRIEREME proposes various optimizations to reduce the time needed to run the SMT solver. These optimizations enable the reuse of results from earlier concolic executions to prune useless solver queries. To show their effectiveness, we collected the median number of solver queries performed per concolic execution in Table 1.

Since the QSYM-backend used by SymCC does not collect statistics on the number of SMT queries performed, we compare the data we collected for our Linear configuration to those collected for our Trie configuration. Our Linear configuration adopts the same optimizations as the Baseline configuration, making them roughly equivalent in the number of SMT queries.

	Queries per exec. (#)		
	Linear	Trie	Δ Queries
`curl_curl_fuzzer_ht...`	75(3)	72(4)	−4.00%
`freetype2_ftfuzzer`	16(5)	9(1)	−43.75%
`harfbuzz_hb-shape-fuz...`	4	4	+0.00%
`libjpeg-turbo_libjpeg...*`	3	9(4)	+191.67%
`libpng_libpng_read_...`	126(3)	120(2)	−4.76%
`libxml2_xml`	8	8	+0.00%
`mbedtls_fuzz_dtlscli...*`	102(6)	181(10)	+78.33%
`openssl_x509`	204(2)	161(1)	−21.27%
`openthread_ip6-send-f...`	88(11)	66(3)	−25.28%
`proj4-2017-08-14`	9	9	+0.00%
`re2-2014-12-09`	3	2	−33.33%
`vorbis_decode_fuzzer*`	150(4)	84(7)	−43.48%
`woff2-2016-05-06`	109(6)	85(7)	−21.79%
`zlib_zlib_uncompress...`	2	1	−50.00%

Table 1: Number of SMT queries per execution. The table shows median and median absolute deviation among trials. Values in each trial are aggregated using their median. Highlighted numbers are statistically significant (p-value < 0.05).

Thanks to the optimizations listed in Section 4.3, TRIEREME in its Trie configuration reduces the number of queries performed in 8 benchmarks in our test suite. The reduction is statistically significant. Note that several benchmarks exhibit a minimal number of queries even in our Linear configuration. This is likely to be a symptom of operations that force SymCC, on which TRIEREME is based, to concretize symbolic expressions, such as uninstrumented library calls. In these cases, it is more difficult for TRIEREME to reduce the number of queries, but it still succeeds with `zlib` and `re2`. Finally, for two benchmarks, `libjpeg` and `mbedtls`, the number of queries is higher for our Trie configuration, but in these cases, this is a good thing. Analyzing the cases, we realized that the behavior is due to our dynamic timeouts. Both benchmarks are very slow and tend to time out often. The timeouts happen due to spending much time in the SMT solver. The dynamic timeouts we added in our Trie configuration allow TRIEREME to significantly reduce the time spent in the SMT solver, stopping queries that are likely to timeout early, thus allowing the concolic engine to perform more queries in the same amount of time.

The other main technique we use to reduce the time spent in the SMT solver is incremental solving. To use incremental solving efficiently, we need path constraints that share a long common prefix. If this were not the case, running incremental queries would perform similarly to running them non-incrementally, as an inline concolic execution engine would do. In order to prove that path constraints indeed share a long common prefix, we have collected statistics regarding path constraint lengths in Table 2.

Each time TRIEREME in the Trie configuration runs an SMT query, we have recorded both the length of the complete path constraint after filtering and its length relative to the last intermediate node, thus excluding its common prefix. When using incremental solving, the trie preserves the information about common prefixes, so only the relative length will influence the time the solver runs, producing a decrease of median query time of 50% in our tests (see Table 8).

	Full length	Length no prefix	Δ Length
`curl_curl_fuzzer…`	7	1	−85.71%
`freetype2_ftfuzze…`	80(6)	1	−98.75%
`harfbuzz_hb-shape…`	267(32)	1	−99.63%
`libjpeg-turbo_lib…`*	431(74)	1	−99.77%
`libpng_libpng_re…`	11(1)	1	−90.91%
`libxml2_xml`	22	1	−95.56%
`mbedtls_fuzz_dtl…`*	230(56)	1	−99.57%
`openssl_x509`	98(6)	1	−98.98%
`openthread_ip6-se…`	10(2)	1	−89.47%
`proj4-2017-08-14`	1	1	+0.00%
`re2-2014-12-09`	37(3)	1	−97.30%
`vorbis_decode_fu…`*	45(4)	1	−97.79%
`woff2-2016-05-06`	550(69)	1	−99.82%
`zlib_zlib_uncomp…`	78(35)	1	−98.71%

Table 2: Length of a path constraint per target branch, including or excluding its common prefix. The table shows median and median absolute deviation among trials. Values in each trial are aggregated using their median. Highlighted numbers are statistically significant (p-value < 0.05).

As Table 2 shows, the exclusion of common prefixes leads to an enormous reduction in the number of new branch constraints the SMT solver needs to process. The median relative length is always 1 for all benchmarks, meaning that it is very likely for a path constraint to share every branch constraint apart from its last one with at least one other path constraint. This proves that using the trie data structure produces query schedules suitable for exploiting incremental solving efficiently.

Given the results presented in this section, we can conclude that the trie scheduler does indeed reduce both the length and the complexity of the queries sent to the SMT solver.

6.2 Q2: Solving Time

The main goal of Triereme is to reduce the time spent running the solver and, as a result, the overall time spent processing a single test case. In Table 3, we show statistics about the time spent in each phase of a concolic execution: apart from showing the total time taken to process a test case, we also divide the execution between time spent inside the SMT solver, generating new candidate test cases, and outside, executing the pipeline described in Section 4.1.

When looking at the Baseline column, it is clear that executing the SMT solver absorbs a significant portion of the execution, about 59% on average across our benchmarks. This means our objective of reducing solver time to improve overall performance is justified.

Observing the time spent outside the solver for the various fuzzers, we can see that Triereme spends a very similar amount of time in this phase in both the Linear and the Trie configurations, with the exception of `libjpeg` and `openssl`. The only difference in this phase between the two configurations is the manipulation of the trie. Given the similarity in time, we can conclude that the overhead introduced by this operation is usually negligible. Looking now at the time spent outside the solver for the Baseline configuration, we can see how it is very often above the one for Triereme. Given the overhead introduced by the serialization and deserialization of symbolic expressions, we expected Triereme to be performing worse than the Baseline configuration; instead, we conclude that the overhead introduced is so small that it gets lost in the performance difference introduced by the implementation differences between Triereme and the QSYM backend used by SymCC. For example, the Linear configuration relies on the deduplication steps for expressions discussed in Section 4.1.

Considering the time spent in the solver, we can observe a mixed behavior comparing Baseline to Linear. We believe this is, again, due to the expression deduplication, which is likely to produce an advantage on slow benchmarks, but may also introduce performance regressions. The difference between the Trie configuration and the other two is quite significant, with a reduction of up to 97% for `re2`. Apart from the use of incremental solving, which provides an advantage across the whole test suite, the benchmarks that time out very often are likely to benefit significantly from our dynamic timeouts. This benefit manifests for the four benchmarks that, in the Baseline configuration, show a mean total execution time above 45 seconds, which is likely obtained through frequent timeouts at 90 seconds, the timeout value set by QSYM. In all of them, the majority of the execution time is spent in the solver, making the reduction of total time even more significant. The `curl` benchmark exhibits the only regression: both Linear and Trie use more solver time than Baseline. This benchmark is characterized by fast concolic executions and a low amount of time spent in the fuzzer; we attribute these characteristics to the short length of its path constraints, which indicates simple solver queries. As a consequence, caching layers end up adding more overhead than the gains produced, doubling the time spent outside the solver. Incremental solving, with such simple queries, is not able to compensate.

Finally, looking at the total execution times for the three configurations, it is clear how the novel design introduced by Triereme is able to reduce the overall execution time, both compared to the previous state of the art, our Baseline configuration, and to the Linear configuration, with a peak speedup of 15.2x.

6.3 Q3: Coverage

In the previous sections we have shown that Triereme in its Trie configuration is capable of processing test cases faster than the Baseline. However, it is important to verify that this speed increase did not come at the expense of coverage contribution, i.e., that Triereme is faster simply because it eliminated operations that were indeed useful for the hybrid fuzzing process.

Final coverage snapshot. In order to verify this, we report the median number of covered edges at the end of our 23-hour experiments in Table 4. Triereme produces statistically significantly more coverage in 7 benchmarks out of the entire test suite, without statistically significant regressions. Our sole regression in execution time, `curl`, here exhibits a statistically significant coverage increase. This means that the slowdown may have also been caused by additional, slower paths that have been explored and it was not large enough to impact coverage progress. These results prove that Triereme still performs all necessary operations to reach the same coverage obtained by our Baseline configuration.

In addition, we find that, if a configuration covers a higher number of edges, it has covered a superset of the edges explored by

	Total	N.Solv.	Baseline Solver	Total	N.Solv.	Linear Solver	Total	N.Solv.	Trie Solver	Δ Solver	Speedup
curl_cu…	0.35(2) s	0.25(1) s	98(4) ms	0.79(3) s	0.58(2) s	0.20(1) s	0.73(3) s	0.56(2) s	0.16(1) s	+68%	+0.5x
freetype…	27(3) s	6.9(8) s	20(3) s	11(3) s	0.76(13) s	9.9(32) s	1.8(3) s	0.89(16) s	0.90(12) s	−96%	+14.9x
harfbuzz…	6.5(8) s	5.3(5) s	3.7(4) s	1.9(3) s	0.94(24) s	0.91(14) s	1.7(2) s	1.3(1) s	0.44(3) s	−88%	+3.8x
libjpeg-…*	24.7(1) s	5.1(2) s	22.2(2) s	23.7(1) s	0.37(3) s	23.4(1) s	18(1) s	5.9(8) s	12(1) s	−46%	+1.4x
libpng_…	30(1) s	9.3(8) s	20(1) s	21(2) s	0.46(5) s	20(2) s	2.0(1) s	0.47(3) s	1.5(1) s	−93%	+15.2x
libxml2…	7.3(2) s	4.1(1) s	5.0(4) s	0.96(5) s	0.51(3) s	0.43(5) s	1.0(1) s	0.53(3) s	0.47(11) s	−91%	+7.3x
mbedtls…*	54(1) s	6.1(2) s	49(1) s	59(1) s	2.9(3) s	57(1) s	22(2) s	3.5(3) s	19(1) s	−62%	+2.5x
openssl…	46.3(2) s	4.0(1) s	42.3(2) s	67.9(3) s	1.8(1) s	66.1(3) s	21(1) s	3.7(1) s	17.4(4) s	−59%	+2.2x
openthre…	1.4(3) s	0.78(12) s	0.67(25) s	1.4(8) s	0.33(6) s	1.1(8) s	0.75(30) s	0.32(7) s	0.37(18) s	−44%	+1.9x
proj4-20…	0.108(4) s	91(3) ms	14.5(3) ms	49(5) ms	35(4) ms	11(1) ms	51(2) ms	36(2) ms	13(1) ms	−13%	+2.1x
re2-2014…	5.3(6) s	4.4(10) s	2.0(2) s	0.62(8) s	0.36(5) s	0.23(4) s	0.38(4) s	0.32(5) s	55(7) ms	−97%	+14.1x
vorbis_…*	64.5(5) s	26(1) s	38(1) s	16(1) s	11(1) s	5.2(6) s	14(1) s	11(1) s	2.5(1) s	−93%	+4.7x
woff2-20…	62(3) s	26(3) s	34(4) s	41(3) s	2.9(3) s	38(2) s	8.4(10) s	3.7(3) s	4.6(6) s	−86%	+7.4x
zlib_zl…	1.2(2) s	0.93(9) s	0.43(12) s	0.37(3) s	0.10(1) s	0.26(4) s	0.15(2) s	97(7) ms	54(9) ms	−87%	+7.5x

Table 3: Total, non-solver and solver time per concolic execution. The table shows median and median absolute deviation among trials. Values in each trial are aggregated using their mean. Variation and speedup are calculated between our Trie and our Baseline configuration, highlighted numbers are stat. significant (p-value < 0.05). The speedup refers to the total time.

				Edges covered (#)
	AFL++	Baseline	Linear	Trie
curl_cur…	10 150(49)	10 205(52)	10 240(33)	10 285(48)
freetype2…	9435(172)	9982(341)	10 058(364)	10 761(283)
harfbuzz…	4782(32)	4813(65)	4834(61)	4807(48)
libjpeg-t…*	3073(3)	3025(8)	2962(32)	3020(14)
libpng_l…	1968(6)	1983(13)	1981(5)	1998(6)
libxml2_…	15 334(73)	15 378(46)	15 599(44)	15 544(50)
mbedtls_…*	2731(18)	2726(16)	2737(20)	3239(319)
openssl_…	5820(4)	5822(4)	5815(4)	5825(3)
openthrea…	2694(180)	2888(281)	2960(317)	2914(298)
proj4-201…	3374(94)	3366(73)	3280(126)	3358(136)
re2-2014-…	2486(5)	2484(4)	2486(3)	2485(4)
vorbis_d…*	1264(2)	1246(5)	1260(2)	1259(2)
woff2-201…	1160(8)	1134(13)	1139(8)	1146(8)
zlib_zli…	457(1)	456(2)	457(3)	456(1)

Table 4: Edges covered at the end of our 23 hours experiments. The table shows median and median absolute deviation among trials. Values in each trial are aggregated using their median. Highlighted numbers are statistically significantly better than Baseline (p-value < 0.05).

the others, with very few exceptions. This is due to the fact that TRIEREME only increases the speed of the exploration, but does not modify SMT queries. This proves that our statistics are collected on equivalent test cases.

Coverage progress. To further explore the performance improvement introduced by TRIEREME, we present a selection of coverage over time plots in Figure 4; the complete set of plots is in Figure 8. During a fuzzing campaign, coverage over time progresses with a logarithmic-like plot, as is evident in this figure. This means that most edges are covered at the beginning of the run, and that coverage eventually reaches a plateau. At this point, the fuzzer makes no meaningful progress on coverage because either the reachable code has been fully explored or the tools used in the fuzzing campaign are extremely unlikely to produce test cases that can progress in the exploration. The plateau can be reached either very quickly, as it is the case for `re2`, or not even in 23 hours, as for `freetype2`. When increasing the processing speed of a concolic execution engine, but not its ability to explore additional code, as is the case for TRIEREME, one can expect to observe faster coverage progress even if it converges on the same plateau. This means that, in cases such as `harfbuzz`, the processing speed increase still shows in the first 8 hours of the process, where the Trie plot is clearly above the Baseline and the Linear ones, even if it is not evident in the final coverage snapshot provided in Table 4.

To aid the visualization of the statistical significance of the difference between the Baseline and the Trie configurations throughout the 23-hour experiments, we show the results of a series Mann-Whitney U-tests, taken at intervals of 30 minutes, in Figure 5. These plots show that only 3 benchmarks, out of the entire test suite, do not exhibit a statistically significant coverage improvement during the 23-hour experiments. When the intervals of statistical significance end before the 23 hours mark, the various configurations are converging toward a single coverage plateau.

Given the final coverage and the statistical significance of the coverage increases throughout the experiments, we conclude that TRIEREME in its Trie configuration has a positive effect on the overall coverage progress of its hybrid fuzzer.

Fuzzing comparison. Finally, we included the AFL++ configuration, which runs just the fuzzer, in Figure 4. Importantly, this configuration progresses faster than the others in 6 benchmarks, e.g. in `harfbuzz`. We expected such behavior because the fuzzer, alone, is able to saturate the portions of code that are easily explorable because it can produce test cases faster. At the beginning of the run, the concolic execution engine effectively takes away resources from the fuzzer, which, at that point, is operating more efficiently. Even in this case, though, the Trie configuration is able to surpass the Baseline by taking away fewer resources due to its

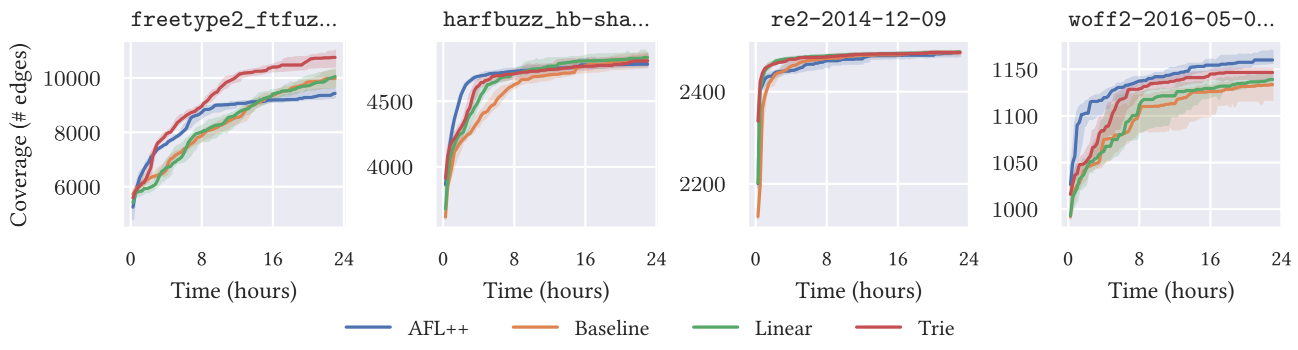


Figure 4: A selection of the coverage plots obtained from our FuzzBench evaluation. The line plots show the median value among trials with a 95% confidence interval.

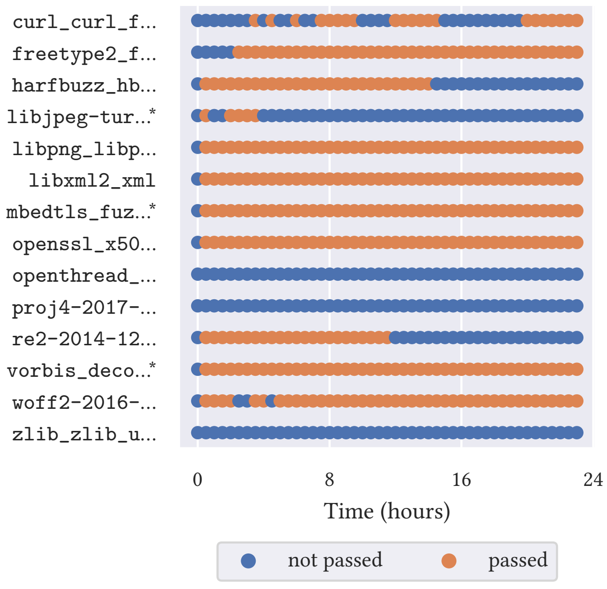


Figure 5: Mann-Whitney U-tests taken at intervals of 30 minutes between our Baseline and our Trie configuration. The *passed* value indicates that the p-value < 0.05 and that Trie has discovered more branches than Baseline.

improved efficiency. Concolic engines are more likely to be useful later in the run, when the fuzzer starts getting stuck on difficult conditions, allowing them to catch it and surpass it. This behavior is visible in freetype2. Overall, TRIEREME performs statistically significantly better than AFL++ in 50% of the benchmarks after 23 hours, while it performs statistically significantly worse in 21% of them. This indicates that, while TRIEREME makes a substantial step in improving the efficiency of concolic engines, some benchmarks are intrinsically less amenable for improvements through concolic execution. Looking at woff2, for example, the concolic execution engine appears to be simply stealing resources from the fuzzer. Given these results and the additional resource usage that comes with concolic execution, we believe that TRIEREME reduces the number of benchmarks in which concolic execution hurts performance, but it is still not sufficient for every target.

6.4 Q4: Dedicated Cores

When solving a specific path constraint, a concolic engine will determine its solution space and then extract a satisfying test case. While it may be possible to extract multiple test cases from the solution space, the fuzzer can typically mutate the test case more efficiently. For this reason, SymCC queries the solver only once, extracting a single solution, and leaves further mutation to the fuzzer. This entails that the concolic engine remains operational only as long as the fuzzer discovers new test cases. When the fuzzer is close to its coverage plateau, the concolic engine will finish processing the test cases produced at the beginning of the run and then idle waiting for new test cases, rarely resuming its execution.

Our single core experiments confirm this across our test suite. After a period of activity, the concolic engine process sleeps for most of the remaining time. Table 5 shows that our Trie configuration is able to idle statistically significantly longer than our Baseline configuration in 79% of the benchmarks, freeing up computational resources that can be used by the fuzzer, as they share a single core.

Previous work [24, 32] used a different evaluation setup, dedicating a separate core to each component of the hybrid fuzzer. We believe that this setup is not suitable for evaluations where the concolic engine frequently idles for a long time. When this happens, the core assigned to it remains unused; as a result, a faster engine will waste more cycles because it will idle earlier, while a slower engine will exploit the core better and thus reduce its overhead.

Nevertheless, we have collected coverage data with dedicated cores to allow comparison with previous work on an equal footing. We conducted such trials on our local machines, as they are not suitable for the standard cloud configuration. We omitted benchmarks with idle time >95% for all configurations, as they essentially use only one core and would thus produce the same results. Table 6 shows that, despite the difference in speed, Trie obtains statistically significant gains in only 30% of benchmarks at the end of the experiment; in terms of coverage progression, Trie performs better in 60% of benchmarks, but for a considerably shorter time compared to our default configuration (see Figures 6 and 7). These findings

	Idle time			Δ %
	Baseline	Linear	Trie	
curl_curl_…	96.24%	96.46%	96.66%	+0.21%
freetype2_f…	0.12%	0.04%	63.48(4)%	+63.43%
harfbuzz_hb…	36.75(7)%	79.58(4)%	82.69(2)%	+3.11%
libjpeg-turb…*	0.17%	0.16%	0.21%	+0.05%
libpng_libp…	32.05(2)%	50.17(5)%	95.09%	+44.92%
libxml2_xml	0.18%	80.61(1)%	79.98(3)%	−0.64%
mbedtls_fuz…*	44.52(2)%	40.22(2)%	75.05(2)%	+34.83%
openssl_x50…	0.29%	1.88%	48.78(1)%	+46.90%
openthread_…	98.17(1)%	98.26(1)%	99.22%	+0.96%
proj4-2017-0…	96.94%	99.85%	99.84%	−0.01%
re2-2014-12-…	81.21(2)%	97.77%	98.66%	+0.88%
vorbis_deco…*	0.18%	67.26(2)%	72.24(2)%	+4.98%
woff2-2016-0…	4.41(4)%	29.06(4)%	84.57(2)%	+55.51%
zlib_zlib_…	99.17%	99.80%	99.91%	+0.11%

Table 5: Percentage of time each concolic engine spends idling. The table shows median and median absolute deviation among trials. Values in each trial are aggregated using their median. Highlighted numbers are statistically significant (p-value < 0.05).

			Edges covered (#)
	Baseline (2 cores)	Linear (2 cores)	Trie (2 cores)
freetype2…*	11 260(271)	11 197(170)	11 268(430)
harfbuzz_…*	10 828(39)	10 867(24)	10 868(24)
libjpeg-tu…*	3074(1)	3073(2)	3074(1)
libpng_li…*	2004(2)	2004(6)	2002(6)
libxml2_x…*	15 604(33)	15 616(20)	15 608(34)
mbedtls_f…*	2760(41)	2755(20)	3584(167)
openssl_x…*	5830(4)	5826(8)	5832(2)
re2-2014-1…*	2870(2)	2867(4)	2872(3)
vorbis_de…*	1260(2)	1262(2)	1262(2)
woff2-2016…*	1152(9)	1152(10)	1140(9)

Table 6: Edges covered at the end of our 23 hours experiments with dedicated cores. The table shows median and median absolute deviation among trials. Values in each trial are aggregated using their median. Highlighted numbers are stat. significantly better than Baseline (p-value < 0.05).

show that dedicating a core per component hides the overhead of slower concolic engines; we thus advise using one core per fuzzer.

7 RELATED WORK

Fuzzing has been recently gaining momentum in the community [3, 14, 20, 30] and since Driller [27] first proposed to combine traditional fuzzing with concolic execution, a rich literature on hybrid fuzzing has emerged. We will discuss the most relevant works and then focus on the relatively fewer works that specifically aim to speed up constraint solving using constraint caching.

Hybrid fuzzing. Driller [27], DigFuzz [34], and Hybrid Concolic Testing [21] selectively apply concolic execution when it is most effective in increasing the efficiency of the overall hybrid fuzzing process. In contrast, our work focuses on reducing solving time regardless of where and when concolic execution is used. These techniques are orthogonal to our approach, and could be combined with our work to achieve even better fuzzing performance.

The most recent works focusing on the performance of concolic execution engines in the context of hybrid fuzzing are QSYM [32], SymCC [24], SymQEMU [25], and SymSan [8]. Among these, only QSYM focused on reducing solving time, while the others focus on the management of symbolic expressions. Our work reduces solving time building upon the techniques introduced by QSYM, but remodelling the solving pipeline to enable optimizations, such as incremental solving, that would be impossible otherwise.

JigSaw [9], JFS [19], and Fuzzolic [6] are recent attempts to side-step full-blown SMT solving by combining various fuzzing techniques to approximate solutions to SMT problems. While our work also makes constraint solving more efficient, it delivers exact solutions and does not use approximations.

KLEE [7] implements constraint filtering and complex caching among other techniques. Our work explores these techniques in the context of hybrid fuzzing, where KLEE's more sophisticated techniques would be too expensive. We show that even simple techniques are effective without introducing too much overhead.

Trie in constraint solving. Outside of hybrid fuzzing, several other works have used a trie data structure to speed up constraint solving. Memoise [31] uses a trie structure to cache a tree of symbolic execution states. Our prototype uses the trie to schedule the solver more efficiently. Green [29] applies constraint filtering on top of other transformations for improved caching efficiency. GreenTrie [17] extends Green by considering the implication relation between (sub-)constraints and stores it using an L-Trie, a trie-like data structure. In contrast, we use the trie structure to store a successor relation between constraints along the execution path. Our use of the trie is sufficiently lightweight to be used in hybrid fuzzing.

Taljaard et al. [28] re-evaluate constraint caching techniques in concolic execution. The authors conclude that the key to caching performance is constraint simplification and that "Z3's incremental mode often outperforms caching". Crucially, TRIEREME's decoupled design allows us to apply all three techniques together.

Other works [16, 26] decouple tracing and solving, but TRIEREME is the first to use the decoupling for optimization in hybrid fuzzing.

8 CONCLUSION

In this paper, we described a new approach to speed up concolic execution in hybrid fuzzers, by (a) decoupling concolic tracing from concolic solving, and (b) scheduling an optimized set of queries to the SMT solver by organizing them in a trie. The trie allows us to reorder and prune queries while allowing the use of incremental solving. As result, we reduce the time spent solving and, consequently, the overall overhead of the concolic engine. Our evaluation proved that our optimizations can lead to statistically significant increases in coverage compared to a state-of-the-art hybrid fuzzer as well as reduce inefficiency when considering a non-hybrid fuzzer.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their feedback and the FuzzBench project. This work was supported by EZK through AVR "Memo" and by NWO through "INTERSECT" and "Vulcan".

REFERENCES

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*, Vol. 19. 1–15.

[2] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1083–1094. https://doi.org/10.1145/2568225.2568293

[3] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful Greybox Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3255–3272. https://www.usenix.org/conference/usenixsecurity22/presentation/ba

[4] Ferenc Bodon and Lajos Rónyai. 2003. Trie: an alternative data structure for data mining algorithms. *Mathematical and Computer Modelling* 38, 7-9 (2003), 739–751. Publisher: Elsevier.

[5] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1621–1633. https://doi.org/10.1145/3510003.3510230

[6] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. https://doi.org/10.1109/ICSE43902.2021.00071

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 209–224.

[8] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. 2022. {SYMSAN}: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2531–2548.

[9] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. JIGSAW: Efficient and Scalable Path Constraints Fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP'22)*. 18–35.

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3 (2011), 265–278.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 337–340.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[13] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS)* (Los Angeles, U.S.A.) *(CCS '22)*. ACM.

[14] Elia Geretto, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2022. Snappy: Efficient Fuzzing with Adaptive and Mutable Snapshots. In *Proceedings of the 38th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 375–387. https://doi.org/10.1145/3564625.3564639

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

[16] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44. Publisher: ACM New York, NY, USA.

[17] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 177–187.

[18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[19] Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. 2019. Just Fuzz It: Solving Floating-Point Constraints Using Coverage-Guided Fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. Association for Computing Machinery.

[20] Stephan Lipp, Daniel Elsner, Thomas Hutzelmann, Sebastian Banescu, Alexander Pretschner, and Marcel Böhme. 2022. FuzzTastic: A Fine-Grained, Fuzzer-Agnostic Coverage Analyzer. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 75–79. https://doi.org/10.1145/3510454.3516847

[21] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 416–426.

[22] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[23] Xianya Mi, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2021. LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating. In *24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID '21)*. Association for Computing Machinery, New York, NY, USA, 62–77. https://doi.org/10.1145/3471621.3471852

[24] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[25] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*.

[26] Emil Rakadjiev, Taku Shimosawa, Hiroshi Mine, and Satoshi Oshima. 2015. Parallel SMT Solving and Concurrent Symbolic Execution. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 3. 17–26. https://doi.org/10.1109/Trustcom.2015.608

[27] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, Vol. 16. 1–16.

[28] Jan Taljaard, Jaco Geldenhuys, and Willem Visser. 2020. Constraint Caching Revisited. In *NASA Formal Methods*, Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou (Eds.). 251–266.

[29] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Association for Computing Machinery.

[30] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1634–1645. https://doi.org/10.1145/3510003.3510174

[31] Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 144–154.

[32] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[33] Michał Zalewski. 2013. American fuzzy lop.

[34] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing.. In *NDSS*.

A APPENDIX

Memory Usage. Table 7 shows the highest memory usage we have observed, per benchmark, in any of our trials. The memory usage reported by our Rust components are fairly similar across our Linear and our Trie configuration. This is because both use the same constraint processing pipeline and caching policies, leading to the same peaks in memory usage. The trie data structure in our scheduler is not, in itself, very large as it contains only references to larger objects in the caches. Since these references keep more cached objects alive, though, we expect a higher average memory usage in our Trie configuration. We can observe that there are three benchmarks that show an increased overall memory consumption (> 4 GB): `libjpeg`, `mbedtls`, and `vorbis`. Even in these

	Rust mem. usage (MB)		Z3 mem. usage (MB)	
	Linear	Trie	Linear	Trie
curl_curl_f...	193.5	196.5	25.1	54.6
freetype2_ft...	736.7	1071.3	830.5	1624.0
harfbuzz_hb-...	357.4	330.9	421.3	933.1
libjpeg-turbo...*	438.6	279.7	1059.1	8621.6
libpng_libpn...	64.7	70.5	44.5	152.3
libxml2_xml	1149.8	1018.3	291.5	1012.0
mbedtls_fuzz...*	3829.0	4334.3	58.0	516.2
openssl_x509	159.1	401.6	161.2	2416.5
openthread_i...	52.5	52.1	46.9	147.9
proj4-2017-08...	188.2	214.9	0.4	3.0
re2-2014-12-0...	71.9	72.5	29.1	49.4
vorbis_decod...*	295.6	277.7	265.5	8378.9
woff2-2016-05...	246.6	263.1	85.5	304.5
zlib_zlib_u...	41.3	44.8	100.2	351.5

Table 7: Maximum memory usage per benchmark obtained in any of the trials in our evaluation. The data was collected sampling memory usage every 3 seconds.

	SMT query time		Δ Query time
	Linear	Trie	
curl_curl_fuzzer_...	1.6(2) ms	1.2(4) ms	−29.13%
freetype2_ftfuzzer	1.9(3) ms	0.79(7) ms	−58.07%
harfbuzz_hb-shape-f...	2.5(1) ms	1.04(3) ms	−57.70%
libjpeg-turbo_libjp...*	3.7(6) ms	0.78(4) ms	−78.89%
libpng_libpng_read...	1.3(1) ms	0.64(3) ms	−48.89%
libxml2_xml	1.2(1) ms	0.82(2) ms	−32.21%
mbedtls_fuzz_dtlsc...*	2.4(3) ms	1.04(4) ms	−56.16%
openssl_x509	2.2(1) ms	1.17(2) ms	−46.38%
openthread_ip6-send...	1.1(1) ms	0.78(10) ms	−30.29%
proj4-2017-08-14	0.84(9) ms	0.69(2) ms	−18.64%
re2-2014-12-09	1.2(2) ms	0.68(9) ms	−41.97%
vorbis_decode_fuzz...*	1.3(1) ms	0.85(8) ms	−32.45%
woff2-2016-05-06	7.8(8) ms	1.1(2) ms	−86.44%
zlib_zlib_uncompre...	3.0(15) ms	0.69(7) ms	−77.36%

Table 8: Time taken to perform a single SMT solver query. The table shows median and median absolute deviation among trials. Values in each trial are aggregated using their median. Highlighted numbers are statistically significant (p-value < 0.05).

cases, though, the memory consumption peaks at 8.7 GB, proving that TRIEREME uses RAM in a practical, although occasionally intensive, way. We believe that adopting more aggressive pruning policies would bring down the usage even more, but we have not explored this possibility in the current experiments.

Query time. Table 8 shows the time taken to perform a solver query across the benchmarks we selected. The use of incremental solving in our Trie configuration is able to substantially reduce the time taken to get a solution for a query. We obtain the highest improvements with the benchmarks that produce the slowest queries, e.g. woff2 and libjpeg. These benchmarks also produce the longest path constraints according to Table 2. For this reason,

	Queries timed out		
	Linear	Trie	Δ %
curl_curl_fuzzer_ht...	0.00%	0.00%	+0.00%
freetype2_ftfuzzer	1.08%	3.19%	+2.11%
harfbuzz_hb-shape-fuz...	0.04%	2.33%	+2.29%
libjpeg-turbo_libjpeg...*	11.54(2)%	4.87(1)%	−6.67%
libpng_libpng_read_...	0.98%	1.89%	+0.91%
libxml2_xml	0.00%	1.47%	+1.47%
mbedtls_fuzz_dtlscli...*	2.18(1)%	0.00%	−2.18%
openssl_x509	0.25%	2.80%	+2.55%
openthread_ip6-send-f...	0.00%	0.19%	+0.19%
proj4-2017-08-14	0.00%	0.00%	+0.00%
re2-2014-12-09	0.00%	1.47(1)%	+1.47%
vorbis_decode_fuzzer*	0.00%	0.60%	+0.60%
woff2-2016-05-06	0.21%	2.78%	+2.57%
zlib_zlib_uncompress...	0.00%	0.00%	+0.00%

Table 9: Percentage of queries that reached the timeout threshold. The table shows median and median absolute deviation among trials. Values in each trial are aggregated using their median. Highlighted numbers are statistically significant (p-value < 0.05).

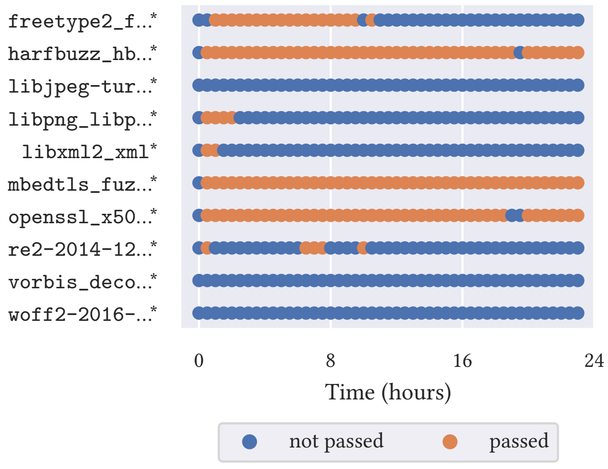


Figure 6: Mann-Whitney U-tests taken at intervals of 30 minutes between our Baseline and our Trie configuration using dedicated cores. The *passed* value indicates that p-value < 0.05 and that Trie has discovered more branches than Baseline.

we can attribute the solving time reduction to incremental solving. Overall, we obtain a mean reduction of 50% in the benchmarks we used for our evaluation.

	Query num.	Path len.	Solver time	Total time	Coverage	Idle time	Cov. (2 cores)	Timeouts	Query time
curl_curl_fuzzer_ht...	1.17e-01	1.14e-07	1.54e-06	1.54e-06	2.26e-02	1.85e-02		1.00e+00	3.02e-02
freetype2_ftfuzzer	6.12e-05	7.89e-09	6.80e-08	6.80e-08	3.29e-05	6.80e-08	6.92e-01	6.80e-08	6.80e-08
harfbuzz_hb-shape-fuz...	6.68e-01	8.01e-09	6.80e-08	6.80e-08	9.57e-01	1.44e-02	4.57e-02	5.37e-08	6.79e-08
libjpeg-turbo_libjpeg...	3.47e-05	2.81e-07	2.64e-05	2.64e-05	5.34e-01	4.62e-01	6.56e-01	1.54e-06	1.54e-06
libpng_libpng_read_...	3.84e-05	6.73e-09	6.80e-08	6.80e-08	1.32e-03	6.80e-08	9.10e-01	2.56e-07	6.79e-08
libxml2_xml	1.51e-03	5.64e-09	6.80e-08	6.80e-08	6.90e-07	2.50e-01	4.29e-01	4.95e-08	4.53e-07
mbedtls_fuzz_dtlscli...	1.53e-06	2.80e-07	1.54e-06	1.54e-06	1.54e-06	1.54e-06	1.54e-06	1.18e-06	1.54e-06
openssl_x509	6.38e-08	7.96e-09	6.80e-08	6.80e-08	6.70e-02	6.80e-08	4.22e-02	6.78e-08	6.78e-08
openthread_ip6-send-f...	6.82e-05	7.59e-09	3.37e-02	2.80e-03	5.52e-01	3.37e-02		6.68e-05	7.57e-04
proj4-2017-08-14	1.00e+00	1.00e+00	1.48e-01	3.94e-07	8.18e-01	4.73e-01		1.00e+00	1.01e-03
re2-2014-12-09	6.99e-04	7.90e-09	6.80e-08	6.80e-08	5.97e-01	1.60e-05	6.62e-01	7.99e-09	2.04e-05
vorbis_decode_fuzzer	1.53e-06	2.77e-07	1.54e-06	1.54e-06	1.56e-06	1.52e-04	1.85e-01	3.92e-07	1.86e-06
woff2-2016-05-06	1.41e-07	7.99e-09	6.80e-08	6.80e-08	5.31e-03	6.80e-08	7.01e-02	6.61e-08	6.80e-08
zlib_zlib_uncompress...	1.51e-03	8.01e-09	6.80e-08	6.80e-08	9.67e-01	6.80e-08		4.51e-03	1.66e-07

Table 10: This table contains the p-values generated by all the Mann-Whitney U tests performed for the other tables in the paper. Values in the other tables are highlighted when the corresponding p-value is lower than 0.05.

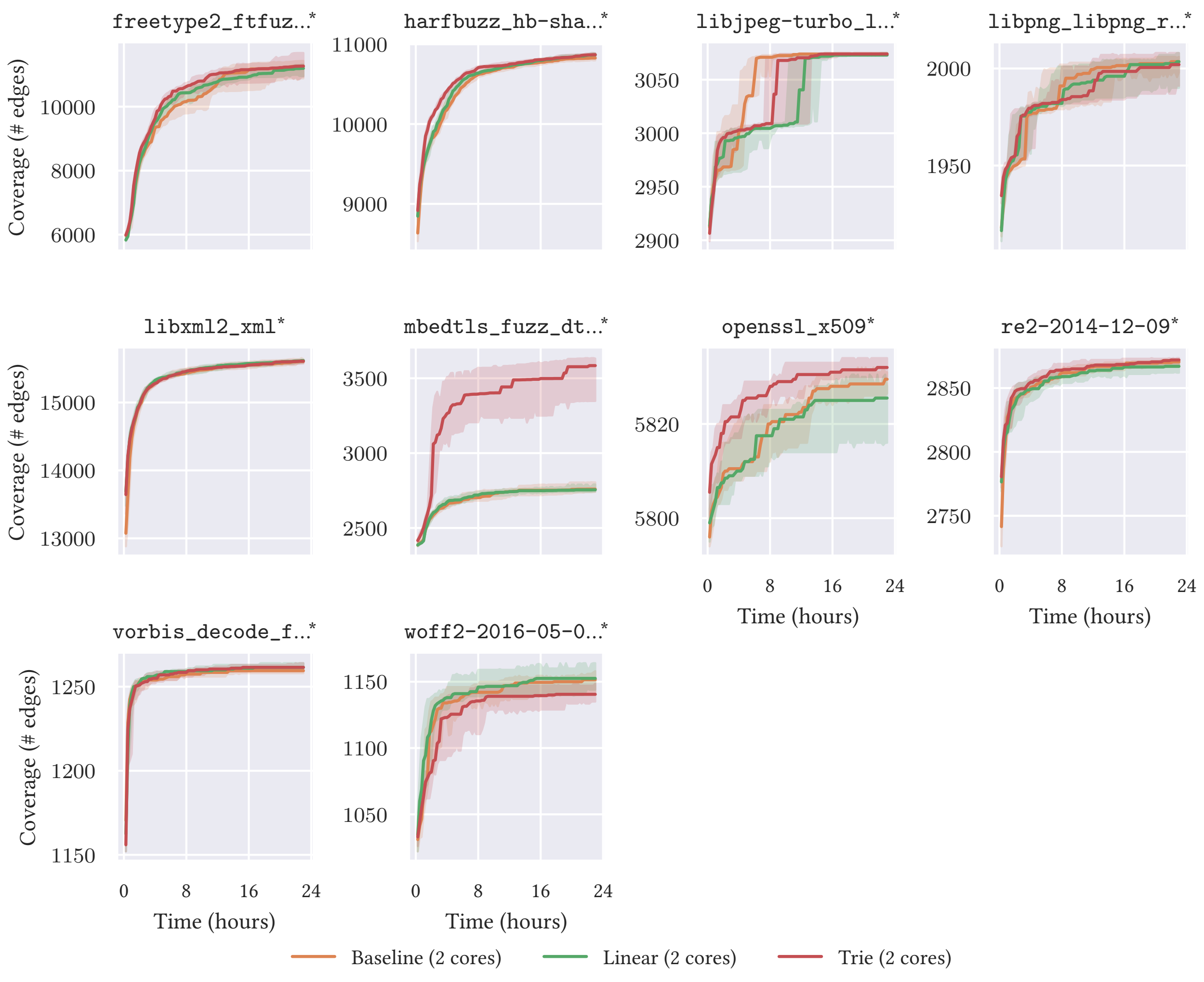


Figure 7: Coverage plots obtained from our FuzzBench evaluation using dedicated cores. The line plots show the median value among trials with a 95% confidence interval.

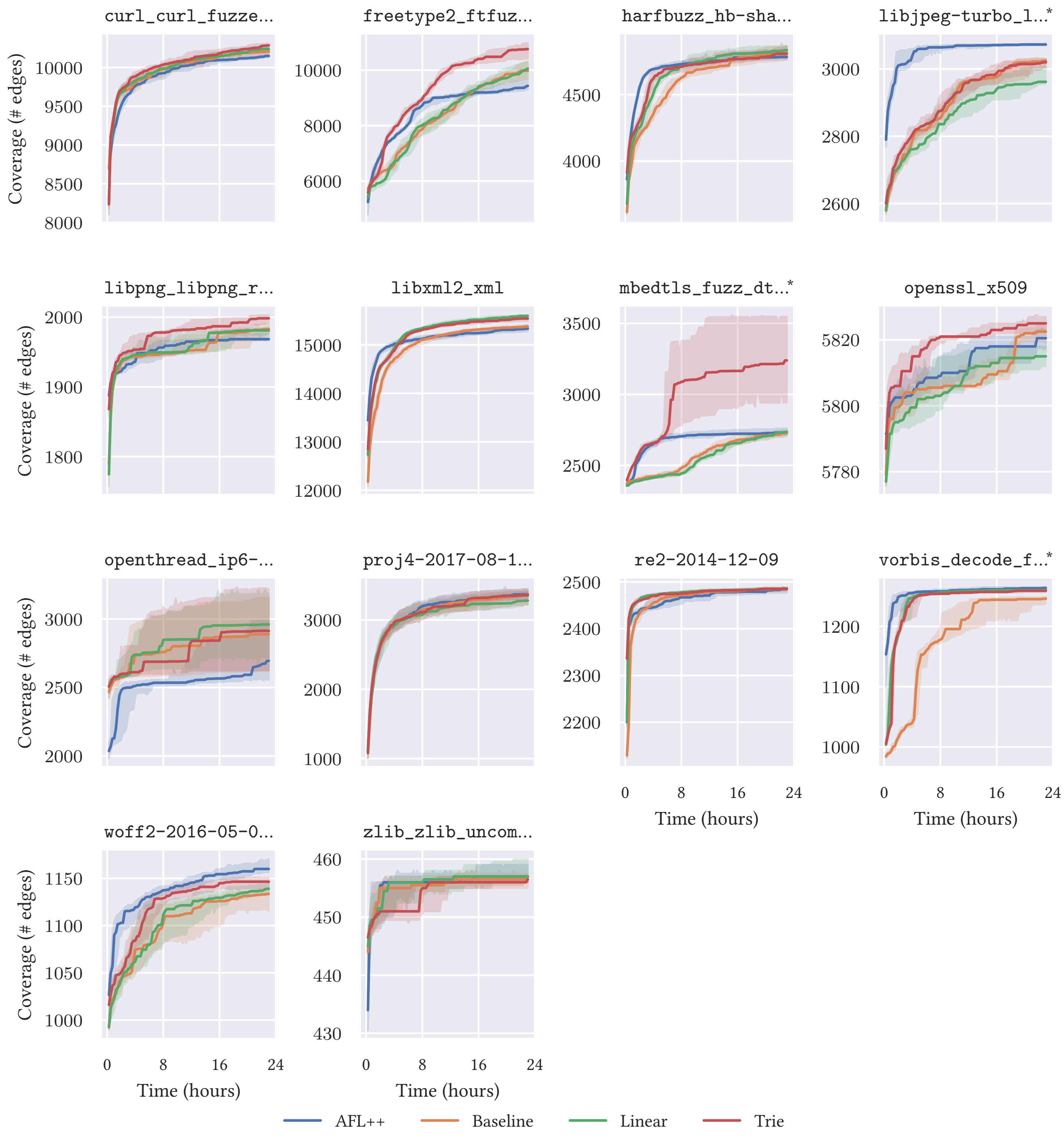


Figure 8: Coverage plots obtained from our FuzzBench evaluation. The line plots show the median value among trials with a 95% confidence interval.