KLEEF Symbolic Execution Engine

Aleksandr Misonizhnik, Sergey Morozov, Vladislav Kalugin, Alexey Babushkin, Yurii Kostyukov, Dmitry Mordvinov

Introduction

We want to present KLEEF [5] — a symbolic execution engine designed for a precise C/C++ program analysis. It has two major modes for two specific kinds of tasks.

The main aim of the first, *error-guided* KLEEF mode is to reproduce error traces, e.g., obtained from some static analyzer. The motivation behind this mode is as follows. Today, using a static analyzer in a product pipeline becomes a norm. However, their users spend a lot of effort and resources on investigating static analysis reports because of a high false positive rate. Software analyzers are usually advertised to be “blazingly fast” but their speed comes at a price of the high false positive rate. After two minutes of static analyzer work, a software engineer can spend entire weeks tracing its reports — and if we add up these times we can no more call the entire process “fast”. Moreover, in most cases manual investigation of these reports ends up concluding that almost all reported traces are false positives. That is, investigating static analysis results requires a lot of time and money, and in the end most results together with all the effort are just thrown away. Thus, the ultimate benefit of KLEEF is that it handles the complex task of investigating static analysis reports and cuts off irreproducible traces. As KLEEF is completely automatic, expensive time of software engineers is not wasted.

Second mode is called *coverage-guided* and its aim is to automatically produce a maximal test coverage for the specified program. Untested code is a security threat because it may be abused by a hacker in an unexpected way. Thus, approaching high test coverage for the production code may greatly decrease security risks. However, getting high test coverage is a hard task for humans, who are bad at tracking complex dependencies and considering all corner cases. This is where KLEEF comes at hand: it generates a maximal test coverage for the program completely automatically. Furthermore, KLEEF puts specific effort to minimize the amount of code in generated tests, so that they could be more human-readable and maintainable.

KLEEF is fine-tuned for both tasks. Consider, for example, a Competition on Software Testing Test-Comp, which has benchmarks for both of these tasks. On these benchmarks KLEEF gains 2360 points, while Test-Comp 2023 winner FuSeBMC gains 2220 and original KLEE gains 1750 points.

Let us now introduce successful *industrial* applications of KLEEF.

1 Industrial Applications

KLEEF already has a number of successful applications in the company. First, let us describe the application for static analysis traces verification and then we will turn to test coverage generation.

1.1 Static Analysis Traces Verification

Consider the trace on a listing 1, which is a simplified version of the trace reported by a real static analyzer used by the company. It was even eye-checked by independent humans and considered to be a true positive, while it is not.

```
 1 typedef struct { unsigned N; } Param;
 2 typedef struct { int magic; } St;
 3
 4 void* Alloc(Param *p) {
 5   void *buf = NULL;      // event 1
 6   Exec(&buf, p);         // event 2
 7   return buf;            // event 4
 8 }
 9
10 void Exec(void **para, Param *p) {
11   if (para == NULL)
12     return;              // event 3
13   *para = alloc(++p->N, sizeof(St));
14 }
15
16 void Init(Param *p) {
17   if (!p) return;
18   St *s = Alloc(p);
19   s->magic = 42;         // event 5
20 }
```

Listing 1. False positive static analysis trace

First event on line 5 is a source: a buffer is assigned to NULL. A pointer to the buffer is then passed to the function Exec on line 6. Then a static analyzer makes a mistake: with the third event it assumes that the function will return on line 12, while it could not. The reason is that para is a pointer to a local variable buf, so para itself could not be NULL, as a condition on line 11 requires. Instead, normal execution of the function will proceed with allocating memory to the buffer on line 13. The buffer is then returned on line 7, and finally line 19 is executed. Following the static analyzer logic, variable buf is unchanged after the call to Exec, so line 19 leads to null pointer dereference. Indeed, as line 12 cannot be actually reached in this trace, the trace report by the static analyzer is a false positive.

Recall that this trace was even checked by a (at least one) human, who also missed the fact that the trace is not feasible. Moreover, the static analyzer which produced this report itself uses a variant of symbolic execution, and it still missed this trace. That is, goofy symbolic execution typically included in static analyzers is not enough to filter out even such simple false positives. Even worse, such symbolic execution can give a false sense of safety, while in practice it may not help filter out nontrivial false positives. It follows that we still need a *precise* and *robust* symbolic execution engine for verification of static analysis reports. We claim that KLEEF is one of those. If we run KLEEF on this program and trace, it successfully handles pointer management and immediately tells that this trace is a false positive as its third event cannot be reached.

Because KLEEF is a well-crafted symbolic execution engine fine-tuned specifically for this complex task, it automatically excluded 65% of traces produced by one static analyzer used by the company.

1.2 Test Coverage Generation

KLEEF is successfully applied for test coverage generation as a backend behind UnitTestBot C/C++ [3].

KLEEF managed to generate a test coverage for an industrial project with tens of millions LOC in 40 hours [3].

Let's study how KLEEF works.

2 How KLEEF Works

KLEEF is developed as a complete overhaul of KLEE [2, 4] symbolic execution engine. First, let us describe how original KLEE works.

2.1 How Original KLEE Works

```
1 void CheckPass(int p, int k) {
2   p ^= 0b110011;
3   r = k & (p >> k);
4   if (r == 0b100)
5     GrantRootAccess();
6   else
7     PrintPasswordFail();
8 }
```

Listing 2. Simple program with a potential security violation

KLEE is a *symbolic execution* [1] engine. Symbolic execution is a generalization of testing. A test involves executing a program on one specific input. For example, if run a function `CheckPass` from the listing 2 with `p = 0b1110000` and `k = 0b100` we will be lucky to get root access to the system. Instead of relying on luck, we could explore the program *systematically* by running it on *symbolic values* of `p` and `k`, i.e., introduce *variables* for their values instead of using some concrete value, like `0b100`.

This is exactly what a symbolic interpreter like KLEE would do. It will start interpretation on line 1 with a symbolic memory mapping $\{p \mapsto \alpha, k \mapsto \gamma\}$, where α and γ are variables, symbolic values. The interpreter then executes the instruction on line 2 and updates the symbolic memory to $\{p \mapsto \alpha \oplus 110011_2, k \mapsto \gamma\}$. Then after line 3 the symbolic memory becomes
$\{p \mapsto \alpha \oplus 110011_2, k \mapsto \gamma, r \mapsto \gamma \mathbin{\&} ((\alpha \oplus 110011_2) \gg \gamma)\}$.

When the symbolic interpreter meets a branching instruction on line 4, unlike a usual *concrete* program execution, it goes to both `if` branches *simultaneously* by forking the symbolic memory. But before proceeding to interpretation of instructions in branches, the interpreter should check their feasibility. To do that, it adds to the initially empty *symbolic path constraint* the branching condition, which for the line 4 would be $\gamma \mathbin{\&} ((\alpha \oplus 110011_2) \gg \gamma) = 100_2$, and pass it to a specific logical solver of such constraints called *SMT-solver*. An SMT-solver checks the formula and if it is satisfiable, returns a model, e.g., $\gamma = 100_2, \alpha = 1110000_2$. A model obtained from the SMT-solver represents a concrete user input which could be used to get to the specific branch of the program, i.e., a test.

If the path constraint together with a current condition is satisfiable, the symbolic interpreter adds the condition to the path constraint and forks to the corresponding branch. If it is not satisfiable, the interpreter does not fork to the corresponding branch. The interpreter thus manipulates with *symbolic states*, which are essentially a combination of a symbolic memory and a path constraint. This method allows one to systematically investigate all feasible program paths completely automatically.

Despite looking simple, symbolic execution is really effective in practice. Yet real life programs pose a number of challenges to a symbolic execution engine, and KLEE is not an exception. Let us now talk about limitations we found while trying to throw industrial code to KLEE and how we overcame them, ending up with KLEEF we have today.

2.2 KLEE Limitations and Our Enhancements

Firstly, we extended the original KLEE with a number of features, which are vital for industrial applications, namely: support for floating point operations, assembly language insertions, undefined behavior sanitizers, LLVM and C++ type systems. More complex enhancements are categorized and described below.

2.2.1 Symbolic Memory.

A lot of symbolic execution engines like KLEE rely on unrealistic assumptions on possible objects behind pointers, which make their application very limited.

We enhanced KLEE with **support for arbitrary data structures** like trees and linked lists by reworking the symbolic memory in KLEE with *lazy initialization*. It is a technique for a systematic exploration of all possible objects in memory to which a pointer could refer. It works as follows. If we need to dereference a symbolic pointer, we fork the symbolic state into many, where each one assumes that the pointer refers to one of possible locations already existing in the memory. We also fork one extra state, where the pointer refers to a fresh lazy initialized symbolic object, which is distinct from all the object from the current symbolic memory.

We further improve lazy initialization with *symcrete* values, which also help to support dynamically allocated arrays (arrays with symbolic sizes) and external calls. KLEEF thus **supports buffer overflows** which are known to be hard to detect for symbolic execution. A symcrete is a pair of *sym*bolic value and its con*crete* instance valid in the current context. When a logical solver receives a query with a symcrete, an equality between the symbolic and concrete parts of the symcrete are added to the query. This helps the solver to solve the query, as a part of the model is already specified in the symcrete. This mechanism helps KLEEF to support dynamically allocated arrays by making both array size and address symcretes. The implementation uses the solver to minimize possible array size and a sparse storage for arrays, so that the entire process does not blow up. For example, KLEEF with symcretes finds 532 defects on Test-Comp buffer overflow related benchmarks, compared to 393 defects found by KLEEF with lazy initialization only (for original KLEE the result is 104).

2.2.2 Extern Calls and Environment.

Another typical source of challenges for symbolic execution in general and original KLEE in particular are external (unspecified in input code) functions, in particular, working with the environment, e.g., printing to screen and reading user input.

We added support for input-output operations and functions from parallel processes to KLEE. We also implemented a number of smart modes for **mocking external functions and global variables**, which differ in balance between analysis speed and precision. Most importantly, we added support for **user function annotations**, which are used and generated by static analyzers. Now, if the static analyzer is run on user-annotated code, KLEEF could check its reports while making the same assumptions the original static analyzer made. For example, despite the fact that the function `alloc` on line 13 in listing 1 is not defined, KLEEF can handle a call to it in a number of ways. If a user specified an annotation for it, KLEEF will use it. Otherwise, KLEEF can be run with a mocking option, in which case it will assume that the function returns a symbolic value, e.g., a symbolic pointer in case of `alloc`.

If an extern function is known and can be executed, KLEEF can handle it via a new **integrated fuzzing solver** based on libAFL. For each query, it generates a LLVM code assuring current constraints and passes it to the fuzzer. The fuzzing solver is seamlessly integrated in the symbolic execution process as extern function arguments and its result are declared as symcretes. This allows KLEEF to soundly work with known extern functions on which original KLEE would give up.

2.2.3 Path Explosion.

One of the most prominent problems of symbolic execution is path explosion. That is, loops and recursion are a source of unbounded growth of the number of symbolic states. On industrial code, it is life-critical to systematically deal with this problem because it is unlikely that the symbolic execution engine will exhaust all obtained symbolic states in any reasonable amount of time.

In order to deal with this problem, we split the KLEE engine into two logical parts. The first part of the engine is a symbolic interpreter, which takes a symbolic state, executes one instruction and produces new states — this part is mostly original KLEE code. The second part is a *searcher*, which chooses a next symbolic state to be executed according to some strategy (e.g., to reach the next location in the trace).

We implemented a couple of searchers from scratch, starting from primitive ones like BFS, DFS and random state-choosing strategies to more advanced guided searchers optimized for specific tasks, like trace verification and test coverage generation. **Trace-guided execution strategy** is based on calculating an approximate distance on an interprocedural control flow graph. It is optimized to choose states reaching trace events based on a prefix tree of input traces. For example, for the trace in listing 1 execution will not fork state at line 17 as we will not event reach the first event in the trace if return on this line.

Coverage-guided strategy can be used with one of seven distinct code coverage criteria: two based on distance to uncovered instructions, two based on instructions covered by the state, two based on depth under branch conditions and one based on recent logical solver query cost.

```
typedef struct {
  int i; int limit; int* status; } Context;

int RunOSTask(Context *ctx) {
  while (ctx->i < ctx->limit)
    ComplexCode(ctx->node[ctx->i++]);
  if (ctx->i == 0) {
    ctx->status = NULL;     // source
    ctx->limit = 0;
  }
  if (ctx->limit != 0)
    return *(ctx->status);  // sink
  return -1;
}
```

Listing 3. A program, which is hard for forward symbolic execution

One of the outstanding features of KLEEF is **bidirectional symbolic execution**. Let us describe it on the example from listing 3. The code illustrates a common pattern in industrial programs: the function performs complex initialization and makes some checks afterward. Note also a typical null-pointer dereference trace from the static analyzer at lines 8 and 12, which is a false positive indeed.

A code written in such a typical way is hard for *forward* symbolic execution, described in the section 2.1. The main reason is that forward symbolic execution, which executes a program in a natural top-down order, will explode while forking at line 5. Each symbolic state after the while loop will proceed to the line 8 and then halt at line 11, as the condition to reach the line 12 cannot be satisfied.

We implemented **backward symbolic execution** step (symbolic execution which interprets the code in the reverse order) to overcome this issue. The implementation is based on executing a code fragment (e.g, lines 8 through 12) in isolation by lazy initialization of program registers. The backward step give us a condition which should be satisfied to reach the target line. For example, to reach the line 12 from the line 11, a symbolic state should satisfy $ctx\text{->}limit \neq 0$. Because of the assignment on the line 9, in order to reach the line 12 from the line 8, the state should satisfy $0 \neq 0$, which is impossible. This fact is saved as a *lemma* in the bidirectional mode of the KLEEF, i.e., the engine proves that the line 12 cannot be reached from the line 8 because these code sections are conflicting. As a result, KLEEF does not execute the expensive while loop at all and quickly shows that the trace is a false positive.

In general, the bidirectional execution works as follows. Firstly, it performs a smart initialization of points for execution in isolation, so that later symbolic execution results could be efficiently reused. On each step, it chooses a next state to be executed with a divergence-preventing complete strategy parameterized by a discrete distribution. When a conflict between some execution points is located, it is minimized using an unsatisfiable core from the logical solver and then added to the incremental database. The approach is sound, as these lemmas are weakest preconditions of given traces. These lemmas boost forward symbolic execution by greatly reducing the number of necessary logical solver calls.

All in all, trace-guided and coverage-guided execution strategies help to overcome the path explosion issue by narrowing the execution to the points of interest. They optimize search when a solution exists, e.g., when a trace is a true positive. The bidirectional execution also boosts this case, but its main power is to deal with problems with no solution, e.g., when a trace is a false positive.

2.2.4 Constraint Solving. Symbolic execution heavily relies on a logical constraint solving. This is the source of symbolic execution strength (its precision), but it is also the source of its weakness. The main weakness of the approach is that a solver is mostly uncontrollable: you just start it and wait. In practice, a solver is run with a limited budget of time, and if time limits, you do not know whether a path is feasible. Thus, you cannot generate a test or show that a trace is a true positive, so the issue cannot be thoughtlessly ignored.

In KLEEF we leave no stone unturned to deal with the issue by calling the solver as rarely as possible. We cache both solver models and unsatisfiability cores, intern symbolic expressions and track constraints during simplification to detect conflicts. On large projects, this helps KLEEF to reduce a number of queries reaching a solver by 99.999976%, which saves an immeasurable amount of time.

When caching does not help, we still have an ace up our sleeve. If we are forced to call the solver, we try hard to call it as smart as possible. For that we implemented a solver wrapper which is fine-tuned to handle symbolic execution specific queries. Note that a symbolic execution investigates a program as a tree, forking at each branching point. Thus, solver queries are not random — they come from different vertices of the same tree. This leads to an observation that most queries share long prefixes of path constraints with other queries, which can be abused by the solver. Our solver wrapper works as a solver scheduler for a fixed pool of real solvers. It calculates a distance between a query and all solvers from the pool and push the query to the solver with the minimum distance. The optimization relies on the ability of logical solvers to work as a stack of constraints with push and pop operations, which helps us not to recompute common prefixes of queries. This optimization helps us gain 10% more points on Test-Comp benchmarks.

3 How to Use KLEEF

Let us now describe how to use KLEEF for static analysis traces verification and test coverage generation.

3.1 Static Analysis Traces Verification

Currently, KLEEF for static analysis traces verification is used via its command-line interface. Yet if your static analyzer is LLVM-based, do not hesitate to contact us, and we will provide a public API, so that you could link your static analyzer against KLEEF. We will now focus on KLEEF command-line interface.

3.1.1 Building KLEEF. Building KLEEF is as easy as installing LLVM compiler infrastructure (version from 6 to 14) and running `build.sh` script[1] in a repository root. It will create a KLEEF binary for you at `~/klee_build/klee_build*/bin/klee`.

3.1.2 Running KLEEF. We collected the best options to call KLEEF here[2] in a style of Visual Studio Code options. We came up with these options from industrial applications of KLEEF for static analysis verification, so you can use them

[1] https://github.com/UnitTestBot/klee/blob/main/build.sh

[2] https://github.com/UnitTestBot/klee/blob/main/configs/options.json

as is. Yet it could be meaningful to tune these options for your domain. The most important options are as follows.

An option `bytecodeFilePath` should be replaced with a LLVM binary `.bc` or `.bca` file you want to analyze.

An option `--max-time` is a time limit, after which KLEEF stops symbolic execution and makes some final postprocessing steps. If you want to guarantee that KLEEF will halt after exactly this amount of time, you can also pass a `--watchdog` option.

An option `--max-forks` says how many times symbolic execution is allowed to fork states on branching instructions. If you decrease this option, KLEEF would halt faster, yet it may not be able to show that a trace is a true positive. The same goes for an option `--max-cycles`, which controls how many times symbolic execution is allowed to enter the same loop.

An option `--analysis-reproduce` specifies a path to a file with a set of traces in SARIF format[3]. You can look for examples of such files across KLEEF tests, like this[4]. Each file contains a set of traces, and each trace has an identifier, a defect type, and a sequence of events, which should be reached to reproduce the trace.

An option `--annotations` specifies a path to a file with functions annotations in the format specified here[5]. You can translate annotations your static analyzer uses to our format so that KLEEF can verify your traces faithfully.

3.1.3 Reading KLEEF output.

For each trace KLEEF reports that it is either true positive, false positive or malformed.

If KLEEF says that a trace is a **true positive**, we are pretty sure it really is, so you can show it to a user.

If KLEEF says that a trace is a **false positive**, it also gives a *confidence rate* in percents, i.e., a confidence of the engine in the verdict. To understand the confidence, imagine the following situation. You run a symbolic execution engine, and it does not halt within a give time limit. You get nothing from this run, in fact, it wasted computing resources for nothing. Now, if you run KLEEF, you will get a confidence rate, which shows, how far it is from an answer to your query. If it is almost 100%, you can increase a time budget for KLEEF a little, and eventually you could get a 100% result, either the trace is verified as true positive or false positive. If the confidence is too low, KLEEF will also communicate an advice which option should be increase. For example, if a trace requires some loop to be traversed ten times, if you run KLEEF with `--max-cycles=1`, it would be not enough. In this situation, KLEEF will report that a trace is a false positive with the low confidence rate and with advice to increase `--max-cycles`.

You can use a confidence rate in a number of ways. You can show a user only traces with a confidence rate lower than some limit, e.g., 50%. You can also show all traces sorted by confidence rate, so that traces which are more probable to contain real bugs will be investigated by the user at first. You can also increase some limits for KLEEF, like time, so that it could confirm more true positives across your traces or increase its confidence.

Finally, if KLEEF says that a trace with a certain identifier is **malformed**, it means that KLEEF was not able to construct a sequence of LLVM blocks for your trace. There could be one of the following problems. It could be that a LLVM binary is not built with debug information, in which case you should rebuild it. It could be that some functions are occasionally missing from a LLVM binary, in which case you should double-check that the binary is built properly. It could be that a location of some event is wrong, e.g., has a wrong line or column.

3.2 Test Coverage Generation

A favorable way to use KLEEF for test coverage generation is via UnitTestBot C/C++, and you can start here[6]. It can be used as a Visual Studio Code or CLion plugin, and even as a GitHub action on your C/C++ project.

Conclusion

We presented KLEEF — our KLEE-based symbolic execution engine for a precise and robust C/C++ analysis. We showed successful industrial applications of KLEEF for static analysis traces verification and test coverage generation. These successes are based on a great number of features we introduced to KLEE to be more impactful on real-life industrial code, leading to its complete overhaul. We briefly described these features and showed how KLEEF can be applied for your own profit.

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).

[2] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[3] Dmitry Ivanov, Alexey Babushkin, Saveliy Grigoryev, Pavel Iatchenii, Vladislav Kalugin, Egor Kichin, Egor Kulikov, Aleksandr Misonizhnik, Dmitry Mordvinov, Sergey Morozov, Olga Naumenko, Alexey Pleshakov, Pavel Ponomarev, Svetlana Shmidt, Alexey Utkin, Vadim Volodin, and Arseniy Volynets. 2023. UnitTestBot: Automated Unit Test Generation for C Code in Integrated Development Environments.

[3] https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html

[4] https://github.com/UnitTestBot/klee/blob/main/test/Industry/test.c.sarif

[5] https://github.com/UnitTestBot/klee/discussions/92#discussioncomment-6242065

[6] https://github.com/UnitTestBot/UTBotCpp/wiki/Intro

In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 380–384. https://doi.org/10.1109/ICSE-Companion58688.2023.00107

[4] The KLEE Team. 2009. *KLEE Symbolic Execution Engine*. http://klee.github.io/

[5] The KLEEF Team. 2021. *KLEEF GitHub Page*. https://github.com/UnitTestBot/klee