# Presentation: Caterpillar: Iterative Concolic Execution for Stateful Programs

Laurent Simon
Samsung Research America
l.simon@samsung.com

Shuying Liang
Samsung Research America
s.liang@samsung.com

Amir Rahmati
Samsung Research America
amir.rahmati@samsung.com

*Abstract*

There is a trend towards running highly-sensitive programs in Trusted Execution Environment (TEE) such as ARM TrustZone or Intel SGX. Such programs typically run in the background and accept requests from less-trusted input, e.g. a less-trusted program may request a sensitive program to generate a cryptographic signature with a key only accessible in the trusted environment. These programs are often stateful. For example, before issuing a signature, a trusted application may require several requests to be issued, e.g. for initialization, to open a session, or to provide a proof of identify. To generate good seeds and test such programs, we propose *Caterpillar*, a system which uses a new concolic approach we call "iterative concolic execution". In this presentation, we will present the principles behind this system and some preliminary results.

## I. Introduction

In this presentation, we consider a special class of programs which we call "stateful programs". We define a stateful program as "a daemon program which accepts requests from clients". Some examples of stateful program are web servers, TLS stacks, and programs running in a Trusted Execution Environment (TEE) such as ARM TrustZone [1], Intel SGX [2], or smart cards (PKCS#11 [3]). For the purpose of this talk, we focus on programs running in TEEs. These stateful programs, although smaller in size compared to other programs in this class, are critical for secure execution of applications such as payment apps, device attestation, and device integrity. These programs provide their clients with certain services that are shielded from less-trusted software. A prime example of such programs is the Android "Key Store", which can be used by apps to generate and store keys, store sensitive data, and perform various cryptographic functions such as signing and signature validation.

The threat model is that in cases where the Android OS is compromised, data stored in the shielded environment should remain secure. Since these stateful programs accept queries from Android kernel and less-trusted Android apps, it is important to ensure that malicious requests cannot take over the stateful program and violate the confidentiality of the sensitive information. Listing 1 provides an example of a stateful program. The program requires several functions to be called in the right order. Each time a function is called, it changes the internal state of the program. Only if the program is in a correct state can the next function go "deep". If we cannot go deep, the main logic of the function cannot be explored by KLEE and the function in step $i$ will return early. Consider $func_1$: it changes the *state* variable to reflect changes made by the function. Then $func_2$ is called: if *state* is not in a good state (i.e. $func_1$ was not called), $func_2$ returns early and no states are explored by KLEE. On the contrary, if *state* is in a good state, then $f_2$ is called and KLEE can explore the main logic of the function. More generally, when calling $func_i$ in step $i$, previous functions in step 1 through $i-1$ must have been called with meaningful parameters so that the *state* changes in a way that will enable good exploration of the next function $func_{i+1}$.

The main contributions of this work are:

- We propose Caterpillar, a system that uses an iterative concolic mode geared at testing "stateful programs".
- We show how to generate useful seeds that go "deep" in the code for this sort of programs. These can then be used for further symbolic exploration with KLEE or as concrete seed for fuzzing, e.g. with AFL.
- We present a preliminary evaluation of the time/state reduction using this method.

```
struct state = {0};

void statefulprog(struct *req, struct *resp) {

  switch(req->cmdId) {
    CMD_1:
      return func1(req, resp);

    CMD_2:
      return func2(req, resp);

    CMD_3:
      return func3(req, resp);

    CMD_4:
      return func4(req, resp);

    // ...
  }

}

int func1(struct *req, struct *resp) {
 f1(&state);
 set_state_f1(&state);
 return OK;
}

int func2(struct *req, struct *resp) {
 if ( !is_state_1(&state) ) return FAIL;
 if ( !f2(&state, req, resp) ) return FAIL;
 set_state_f2(&state);
 return OK;
}

int func3(struct *req, struct *resp) {
 if ( !is_state_f1(&state) ) return FAIL;
 if ( !is_state_f2(&state) ) return FAIL;
```

```
39  if ( !f3(&state, req, resp) ) return FAIL;
40  set_state_f3(&state);
41  return OK;
42 }
43
44 int func4(struct *req, struct *resp) {
45  if ( !is_state_f1(&state) ) return FAIL;
46  if ( !is_state_f2(&state) ) return FAIL;
47  if ( !is_state_f3(&state, req, resp) ) return
        FAIL;
48  if ( !f4(&state, req, resp) ) return FAIL;
49  return OK;
50 }
```

Listing 1. Example of stateful program

## II. Caterpillar

Running a stateful program symbolically through KLEE results in lots of overhead as a lot of time will be spent in states that have exited early in previous functions, and will not lead to exploration of the function's main logic. To alleviate this problem, we use two key ideas:

- First, after exploration of $func_i$ in state $i$, we prune off all states that are unsuccessful. The success of a function is typically encoded in its return value or in a local variable we can test before the function returns. This allows us to reduce state explosion before exploring the next function $func_{i+1}$.
- Second, we concretize the remaining (successful) states, and reuse them as seeds in new runs of KLEE: we start KLEE in replay mode only and switch to symbolic execution at the start of the next function $func_{i+1}$ we want to explore. These seeds put the program under analysis in a good state to explore $func_{i+1}$; and they reduce the strain on the constraint solver as previous functions are run concretely.

## III. Implementation and Evaluation

We implemented Caterpillar using KLEE 3.4 and LLVM 3.4. We use an LLVM pass to insert a call to KLEE's intrinsic *klee_assume* before the return of every function of interest, i.e. $func_i$ in our example above. We made changes to KLEE to be able to switch from "replay mode only" to symbolic execution based on a function of interest given as command line option. We then drive the iterative process described in the previous function. That is, in each step $i$, we use the seeds from step $i-1$, start KLEE in replay mode until we enter $func_i$ at which point we switch to symbolic execution.

## IV. Presentation Format

Our presentation at the KLEE workshop will be divided in the following sections:

1) Overview of Stateful Programs: We will discuss stateful programs, their unique characteristics, and their importance.
2) Iterative Concolic Execution: We will describe the idea behind iterative concolic execution.
3) Implementation: We will present our implementation of Caterpillar. This will include
   - LLVM pass to insert calls to KLEE intrinsics *klee_assume* in the original program.
   - Changes made to KLEE: switch from seed mode to symbolic mode based on function executed.
4) Preliminary Results: We will present some preliminary results we have using the tool after running it on some internal code base:
   - Time reduction vs. naive symbolic execution vs. symbolic execution with state pruning only
   - State reduction vs. naive symbolic execution vs. symbolic execution with state pruning only
   - Bugs found
5) Limitations and Future Directions: We will talk about certain limitation of the approach itself, as well as limiting factors of our implementation:
   - Order of calls to stateful functions
   - Multi-step vs. single-step iteration at the moment
   - Seed mode vs. symbolic is a binary choice in our case: we must launch KLEE with each seed individually. Support for multi-seed where certain states are in seed mode and others in symbolic mode would help. This will hopefully sparks some discussion on the possibility to achieve this.
6) Future Work & Conclusion: We will sum up the current state of this work and its limitations. We hope to start discussion on the possible ways forward to improve this tool.

## References


**[1]** ***ARM TrustZone*. https://www.arm.com/products/security-on-arm/trustzone.**

**[2]** ***Intel SGx*. https://software.intel.com/en-us/sgx.**

**[3]** ***PKCS #11 Cryptographic Token Interface Profiles Version 2.40 OASIS Standard*, Apr. 2015. http://docs.oasis-open.org/pkcs11/pkcs11-profiles/v2.40/os/pkcs11-profiles-v2.40-os.pdf.**