PSA-Level TF Fuzzer Tool

Gary Morrison, Arm Inc.
2020/07/01

TF (PSA) Fuzzing Tool

For PSA-level Directed-Random Testing – Motivation and Historical Background

- There have been reports that one of the PSA-certification labs wants to develop their own in-house proprietary PSA-API fuzzing tool. Ideally, we'd prefer there be an open-sourced such tool to benefit the TF.org community.

- Gary Morrison (Arm Austin) has been working on such a tool, and would be delighted to get it into the open-source world:

  - *It is still in its early stages*, just one guy programming for a few months, but it's beginning to become a useful tool.

  - Initially for TF-M, but conceptually, just switching a single file of "boilerplate" code snippets would be all that's needed to retarget it to TF-A SPCI testing.

TF-M Fuzzer, Originally

- *The original intention* was to provide a program *running on a TF-M target system* that generates TF-M calls based upon a script.
- Its main goals:
  - To *make it easy to write lots of tests* quickly.
  - *Random testing at the PSA level*, with varying levels of pre-determinism vs. randomness.
  - Being able to check for *security breaches*, and to accumulate a *security-regression suite* for TF-M.
- There are complications to running a “test interpreter” on a TF-M target.

TF Fuzzer, for Now, is Workstation-Based a Code Generator

Workstation Code-Generator Advantages:

- Workstation-generated tests require only a very small footprint on the target.
- A test generator, generating "tests like any other," *leverages existing infrastructure better*. An on-target test interpreter would involve "special" scripts – an all-new testing framework.
- It's likely easier to make a *code generator* that can *target PSA calls for both TF-M* ***and TF-A*** than to maintain a "test interpreter" in two ***very-different*** frameworks.

Target-Based Test-Interpreter Advantages:

- At least conceptually, interpreted tests could be downloaded as *data*, whereas generated-executables (typically at least) require more time in flash-image download.
- Some expected-result information is hard to predict "at compile time"; a target-resident agent could even predict correct results when multiple threads vie for common resources.
- There's no single standard interface for FLASH-programming (TF-M at least), which could complicate the automation.

In the Long Run, Co-operative Test Management

There are many ways tests could be managed co-operatively, host and target

- On target systems with fair-sized memory, *its probably possible to run the TF-M Fuzzer on the target itself*, directly performing the calls rather than generating source code. This would generate and run tests *lots of tests very rapidly*.
- It's probably easy enough to provide a "plug-in" mechanism, whereby specialized code can be run in coordination with the general PSA-level exercising. This could be useful, for example, if the Partner adds extensions atop basic PSA itself.

- A RAM- and/or serial-line-based code downloader could be developed to quickly download test templates, or whole generated tests, rather than having to program them into flash.
- A target-based agent, with a small memory and compute footprint, *could re-randomize data and potentially other aspects of a test case* compiled on the workstation, re-running each time.

TF-M Fuzzing Test-Template Language

Test-Templating Language

- A test template describes the *general schema of a test*, with varied determinism vs. randomness.
  - *Completely-random* stimulus is usually not useful; it typically just causes *a lot of errors*.
  - Yes, testing error handling – sometimes called “negative testing” – is valuable too.
  - However, since most code making it to the fuzzing stage doesn’t typically do *a lot of* stupid things, a medium-to-high proportion of the testing needs to operate within the envelope of correct activity.
- It’s usually most interesting to *comparatively-deterministically* setup a test-environment of interest, and then *comparatively randomly* “play around” within that environment.

High-Level Abstraction vs. General-Purpose Language

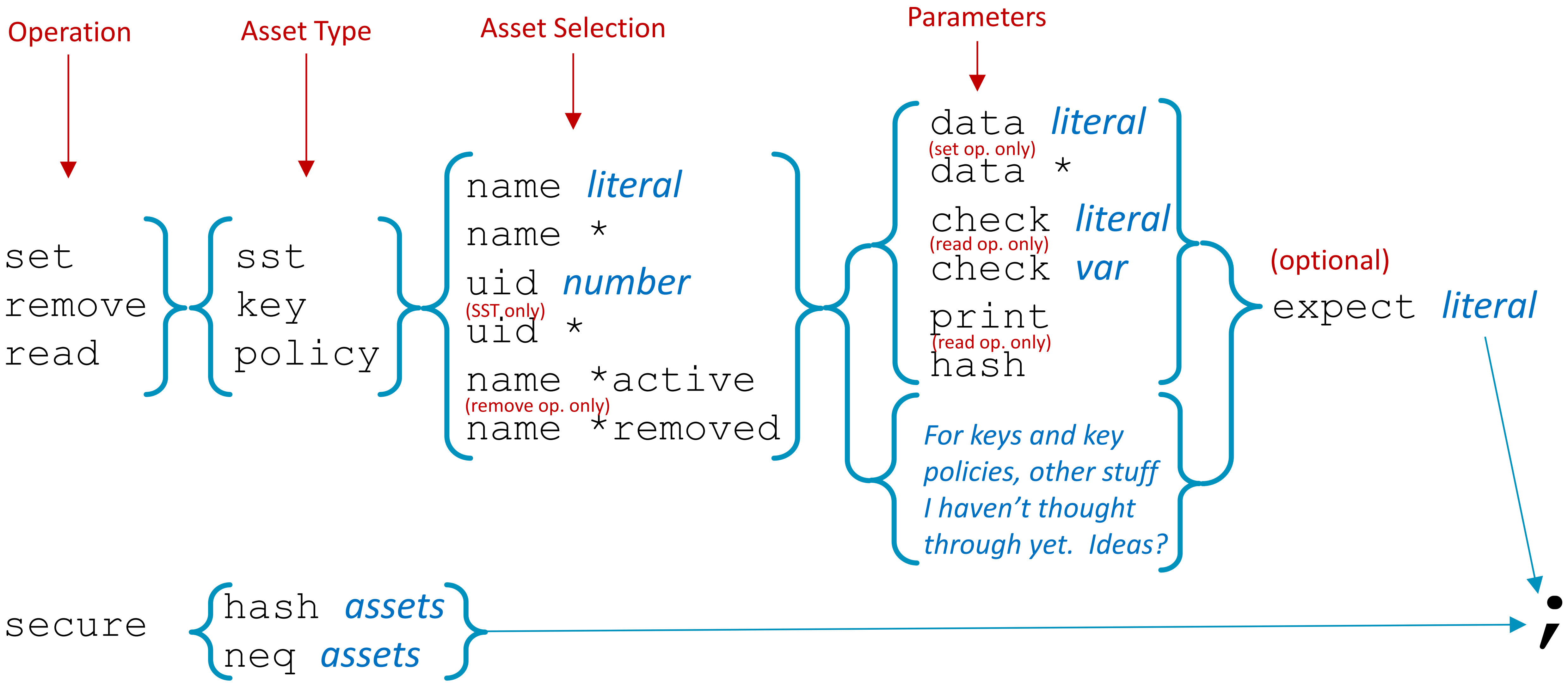
- It’s conceptually possible to put Java or Python (etc.) onto a target.
- The advantage is that *you can do anything you’d ever want*, which is clearly a very powerful capability. There are disadvantages though:
  - A general-purpose, high-level language interpreter ***could*** *take a lot of space on very-small TF-M targets*. Nevertheless, some “embedded-friendly” interpreters (e.g., Micropython) do exist.
  - More importantly though: Although you *can* do everything you want with a general-purpose language, you also ***must*** do everything you want.
- A simplified abstraction also makes it *quick and easy to write lots of tests*!

The Downside of Providing a General-Purpose-Language Interpreter:

How High-Level vs. Low-Level a Language? (ctd.)

- For TF-M target systems with enough resources to support it, adding general high-level language support would be valuable.
- However, let's *first* make it *really easy* to generate *a whole lot of tests* that involve the most important PSA assets and API actions.
- We can then extend that with customized plug-ins.
- Then, we can add full-blown general-purpose language support.
- The following slide briefly summarizes the initial test-templating language the tool currently supports. The demo, shortly, illustrates it, live.

Partial, Quasi-Syntax Chart for Test Templates



Fuzzing Also Requires Modeling

- We also have to remember that, for fuzzing (or at least for “random testing”), once you’ve automated generating random activity, *you’ve still only done half of the job*.
- The other half of the job is *modeling* to generate *expected results*!
- This modeling is extremely hard to do in the most general case, especially if you include multiple interacting asynchronous threads.
- We can’t realistically try to solve this whole problem right from the start. *We should start with a smaller set of use cases and work our way out*.

Demo

Demo Overview:

- What you'll see here is *just a start*, and its "vocabulary" of PSA calls is, for now, small, but we have an easily-extensible framework.
- So far, it can generate asset creation/modification, reading, and removal of SST and Crypto assets.
- One of the goals is just to make writing tests quicker. So, I'll start out with *"party tricks"*: Not amazing capabilities, but just *generating a lot* of testing *from very short templates*.
- Then I'll show TF Fuzzer generating randomized data and asset names.
- Then operating upon multiple assets in a single template line.
- Then I'll demo some test control-flow randomization provisions.
- If you're inside Arm, please look over this confluence site.

Yeah yeah yeah, get on with the demo!

Internals

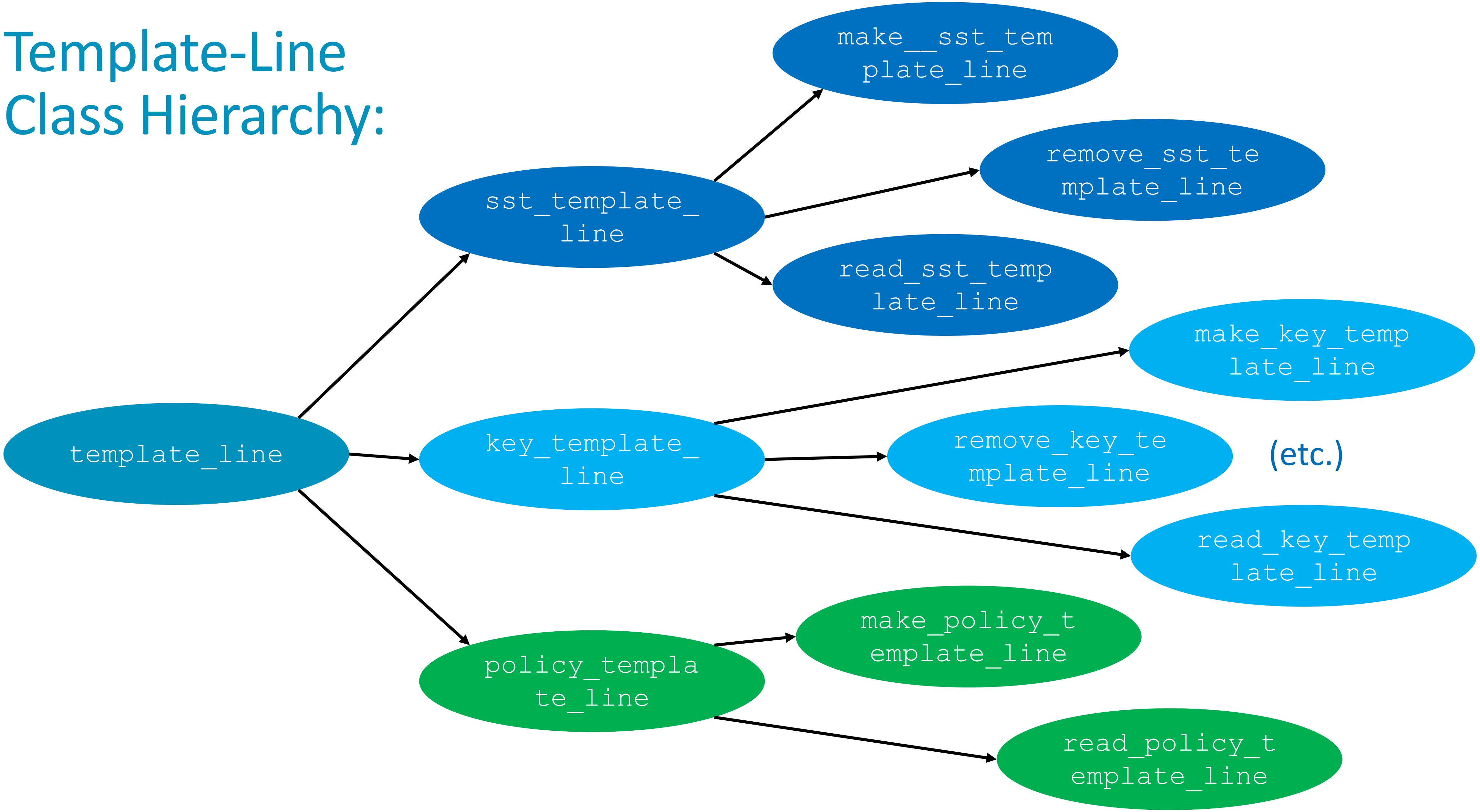
Classes and Tracker Objects

- TF-M Fuzzer uses three main class trees, implementing "tracker objects":
  - Template-file lines:
    - As we parse lines in the template file, they get decoded into these objects.
    - One template-line object per line in the template, although not necessarily only one "alive" at a time.
    - These are `new()`ed once enough is parsed from a template-file line to know which kind to allocate.
    - They are `delete()`ed when no more code needs to be generated from that template line.
  - PSA assets:
    - These track the state of known PSA assets (SST files, Crypto keys, Crypto key policies, etc.)
    - Asset trackers never go away once allocated. If an asset is removed, its tracker is moved onto an STL `vector` of removed objects.
    - Three `vector`s of each asset type are maintained: ***Active*** (present on the system), ***Inactive*** (existed but subsequently removed, and ***Invalid*** (actually, I'm not yet sure what to do with the Invalid list yet!).
    - These three vectors exist for each basic type of asset, so far (again, SST files, Crypto keys, key policies, etc.)

Classes and Tracker Objects (ctd.)

- TF-M Fuzzer uses three main class trees, implementing "tracker objects":
  - PSA calls generated:
    - As it parses the template, it creates a sequential vector of PSA calls to be written out.
    - These call tracker objects include information to create variables used by the calls, as well as the calls and their checking code.
- There's a lot of overlap in types of information in each of these object ***types*** store, but…
  - Their ***lifespans*** are very different (template-line trackers come and go with parsing, whereas asset trackers and call trackers stay throughout the test.
  - Although the information ***types*** are similar, specific information in each can differ!
    - Information in the PSA-asset trackers is ***continually updated*** to reflect the state of that asset as the test progresses.
    - Information in the PSA-call trackers includes a snapshot of relevant information about an asset ***at the time of that call***.

Template-Line Class Hierarchy:



Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধন্যবাদ
תודה