Yash Pandey

GitHub: github.com/EmperorYP7
Website: emperoryp.live
Email: yash.btech.cs19@iiitranchi.ac.in
LinkedIn: linkedin.emperoryp.live
Phone: +91 916726 0712
TimeZone: GMT +5:30
Location: Mumbai, India

Casbin GSoC 2021 Proposal

About me

I am a sophomore at the Indian Institute of Information Technology, Ranchi, India pursuing Bachelor of Technology (Honours) majoring in **Computer Science and Engineering** with a minor in **Software Engineering**. I have been profoundly influenced by technology and the various ways it can be used to express art. My interests lie where technology intersects with art - **Game Development**, **Web Development** (proficient in **front-end**), Music Production, Video Editing and VFX. I have been learning about programming ever since I was in Grade 8. Soon I explored C++ and learned about advance programming through my school teachers. This grew on me even more when I went on for my graduation where I contributed to various open-source projects. This helped me learn more about production ready programming which I tried to implement in my projects as well.

I've contributed to ASFW's OpenColorIO (a color management tool for VFX in C++) CNCF's Layer5 website, Google Flutter's website and small patches to Google IREE and FOSSASIA's open-event-frontend. I am appointed as the co-ordinator of Web and App Development Community of House of Geeks, the technical society of my institute where I work as project maintainer and guide passionate developers of my institute know more and discuss about open-souce software development.

While working with Layer5 and House of Geeks, I learnt a lot about web development, made many projects and explored about GitHub Actions and various approaches for implementing CI/CD. I also have familiarity with **GoLang** and **MERN** stack. I made a CLI based on Cobra for generating templates named TMake. **I've showcased all of my projects here**.

During the first year of graduation, I learnt a lot about C++ by developing a Game Engine which was made using MS Visual Studio, GLFW for target window, ImGui for GUI, and developed a rendering API based on Modern OpenGL. This was inspired by the "Game Engine Series" of TheCherno, a YouTuber. I do have a lot of future plans for this project.

This is the first time I am applying for this program, i.e. I haven't been a GSoC mentee/mentor before.

Technical Skills

- **Programming Languages**
  - Proficient in C++17 and concepts of OOP
  - JavaScript ES6+
  - GoLang *(Basic)*
  - Python *(Basic)*
- **Development Environment**
  - IDE/Editor: Visual Studio Code
  - Visual Studio Community 2019, XCode, CLion
  - Build Systems: CMake, MS Build
  - OS and Architecture: • Darwin arm64 20.3.0 (macOS Big Sur 11.2.2)
    • Windows 10 x64 *(Secondary)*
  - Compiler: Clang 12.0.0, GCC-10, MSVC latest
- **Version Control**
  - Git version 2.24.3 (Apple Git-128)

Meeting with Mentors

I am available anytime between 03:30 to 20:30 hours UTC (09:00 to 02:00 hrs IST) through Email, Gitter and GitHub issues.

I am open to attending video sessions if required.

Abstract

Casbin is an authorization library that extends its features to implement Access Control Lists, Role-Based Access Control, and Attribute-Based Access Control models in various programming languages to its clients. Casbin's Core Engine is written in GoLang. Casbin-CPP has obvious benefits of speed and efficiency compared to its implementation in other languages and thus, benchmarking is vital for the project to stand out from the rest. Python is the most versatile as well as the most used programming language and has huge community support. Casbin-CPP has the potential to support new PyCasbin to compound the benefits of both languages through language bindings and extension libraries. Currently, the project uses Microsoft Unit Testing Framework for C++ for testing and Microsoft's Azure DevOps pipelines for Continuous Integration. CTest is truly cross-platform and can be configured using GitHub Actions for consistent and transparent Continous Integration.

Project Ideas

The projects I propose to work on for GSoC 2021 are as follows:

- Modernizing the project with C++17 standard and Google's benchmarking tool
- Implement Python bindings for Casbin-CPP using pybind11 library
- Implement testing based on CTest and set up workflows for Continuous Integration based on GitHub Actions.

Implementation Details

Primary Goals

Modernizing the project with C++17 standard and Google's benchmarking tool

The project currently uses C++11 as the C++ standard. To keep up with the growing demands and advancements, we need to upgrade to C++17. The direct implication would include using `make_unique` and `make_shared` instead of new, RAII (Resource Acquisition Is Initialization) based mutex locks, wrapping project intrinsic objects with namespaces (since there were many ambiguous symbols on compiling with C++17 as standard), etc. This also includes proper documentation of code.

Benchmarking makes it easier for the developers to understand and track the performance and efficiency of each part and function of a program. Benchmarking has been implemented in casbin's core engine using GoLang's testing module. We can achieve cross-platform, easy-to-use, and similar to GoLang's benchmarking with Google's benchmarking tool. Here's an example of a benchmark test for cached enforcer, say `/tests/benchmark/bench_cached_enforcer.cpp`:

```
#include <benchmark/benchmark.h>

#include "enforcer_cached.h"

static void BM_Enforcer(benchmark::State& state) {
    CachedEnforcer e("basic_model.conf", "basic_policy.csv");
    for(auto _ : state)
        e.Enforce("alice", "data1", "read")
}
```

I have implemented the above benchmarking workflow through CTests here. (only for demonstration)

Implement Python bindings for Casbin-CPP using pybind11 library

Python bindings are simply headers that can be accessed by both the C++ compiler and Python's interpreter as modules. Although we can make bindings without any external library, pybind11 will help us build over many layers of abstraction without bothering/debugging low-level code and yet provide enough flexibility to accommodate every function and class with appropriate inheritance (even smart pointers). This is similar to Boost.Python library. Below is an example for a simple Python binding for a C++ header file, say - `casbin/enforcer.h`:

```
class Enforcer {

public:

    Enforcer(string model_path, string policy_file);

    bool Enforce(vector<string> params);

};
```

Its corresponding binding file will be as follows:

`/casbin/bindings/python/py_enforcer.cpp`

```
#include <pybind11/pybind11.h>

#include “enforcer.h”

namespace py = pybind11;

PYBIND11_MODULE(pycasbin, m) {

    py::class_<Enforcer>(m, "Enforcer")

        .def(py::init<string, string>())

        .def("Enforce", &Enforcer::Enforce);

}
```

There are many ways to build these bindings and to make a python module. The one I would prefer for this is through CMake. This is because Python modules are to be built as a dynamic/shared library in UNIX-based systems and as a static library for Windows with a .pyd extension. This is also in line with the proposal for a new CTest based testing. Also, managing external packages through CMake is well defined and quite customizable. **I've tried to implement the build system for this here**. This can also incorporate builds specific to MS Visual Studio through CMake.

After we successfully build a python module out of `py_enforcer.cpp`, we may call these functions in subsequent Python files likewise:

`enforcer_test.py`

```
import unittest
import pycasbin

class TestEnforcer(unittest.TestCase):

    def test_enforce(self):

        e = pycasbin.Enforcer("./rbac_model.conf",
            "./rbac_policy.csv")

        params = ["alice", "domain1", "data1"]

        self.assertTrue(e.Enforce(params))
```

As trivial as it may seem, working out a proper build systems and testing for these bindings will be a humongous task in light of changing the entire testing framework to CTest. Even if we rely on unit tests on Python, the changes made to testing won't be necessarily mutually exclusive with the entire project. This would require careful refactoring of code and would also include completion of the API to match PyCasbin. The above code would be significantly more efficient than the native Python implementation of Enforcer if we were to scale it up to real-world applications.

Implement testing based on CTest and set up workflows for Continuous Integration based on GitHub Actions.

Testing is an aid to software development. The project currently uses MS Unit Testing Framework for C++. This might take a toll on a substantially large population of developer communities relying on UNIX-based operating systems to replicate these tests locally. Thus, the incorporation of CTest with googletest as a standalone testing framework in the project will result in truly cross-platform and profoundly consistent testing across all operating systems. This will also attract a plethora of open-source contributors without their OS being a hurdle.

Being aware of the Contribution guide, Makefile and MS build is the only way to go as of now. This is mainly to avoid complicated build systems for the library. In fact, Visual Studio natively supports CTest for testing while also giving solid ground to developers who're using UNIX. Google's testing framework is renowned across many opensource projects and communities for its consistency and being lightweight.

Since we're using Google test, here's what a test file will look like, say `/tests/test_enforcer_cached.cpp`

```
#include <gtest/gtest.h>

#include "enforcer_cached.h"

TEST(CachedEnforcerTest, RawEnforceTest) {
    CachedEnforcer e("basic_model.conf", "basic_policy.csv");

    EXPECT_EQ(e.Enforce("alice", "data1", "read"), true);
    EXPECT_EQ(e.Enforce("bob", "data2", "write"), false);
}
```

I tried to implement such a testing framework with google test in this development branch. (only for demonstration)

Secondary Goals

If I happen to achieve all the primary goals for GSoC 2021, I'll be working on secondary goals as well:

- **Initiate integration with mosquitto and envoy proxy**
- **Investigating about more efficient synchronised algorithms using raft**
- **Incorporation of casbin model in native C++**

Casbin-CPP, at the end of the day, is a product. And there's no better way to appreciate the project but to integrate it with various other projects like Mosquitto and CNCF's Envoy Proxy.

Milestones

Milestone 1

I'll be analyzing the code and making relevant changes to be on par with the C++17 standard. This would also include a bit of in-code documentation with comments and completing the API. As suggested by Joey, I'll be implementing a basic watcher for distribution deployment as well. Setting up Google benchmark for the project. I'd also be taking a look at the build systems and configure them for upcoming milestones.

Milestone 2 *(Deliverable before Phase 1 Evaluation)*

I'll start working on Python bindings. This will include developing a thoroughly planned workflow for managing the pybind11 module, ensuring successful build on all platforms, and testing the bindings through CMake. I'll complete all the bindings with tangible tests written in Python along with the build system. This milestone will mark the initial completion or scaffolding of Python bindings for most of the functions and features in the project.

Milestone 3 *(Deliverable before Final Evaluation)*

Refactoring CMake build system throughout the project. CMake will generate solution files for MS Visual Studio on windows. Preparing a consistent single command testing framework based on CTest with Google test. Setting up GitHub Actions to test the library in every platform. Trying to implement a lock-free algorithm for synced enforcer. I'll also be completing the python bindings with tests.

Milestone 4 *(Wishlist/If time permits/post GSoC)*

I'll go ahead and analyze the ways to integrate this project with mosquitto by making abstractions and developing an authz middleware. We'd also be investigating more efficient synchronized algorithms using raft, looking for ways to incorporate casbin model in native C++, and working on MySQL, PostgreSQL adapters.

Detailed Timeline

Present - May 16 (Homework)	Working and learning more about benchmark, pybind11 and CMake build systems
May 17 - June 6	Community bonding period: clarify implementation details with mentors and initial setup
June 7 - June 14	Adhering the code to C++17 standard with comments and completion of API
June 15 - June 22	Setup build system for benchmarking and adding essential benchmarks
June 23 - June 26	Implement watcher for distribution deployment (**Milestone 1**)
June 27 - July 3	Configuring pybind11, workflow for building and creating Python bindings
July 4 - July 12	Adding unittests for the Python bindings
First Evaluation	**Milestone 2 completed**

July 17 - July 23	Revamping build system for the main project, lock-free algorithms for synced enforcer
July 24 - July 30	Setting up test through CTest and google test framework
July 31 - August 5	Adding remaining tests and configuring GitHub Actions for CI/CD, completing python bindings
Milestone 3 completed	
August 6 - August 12	Completing benchmark, tests and python bindings, adding comments wherever necessary
August 13 - August 15	General code cleanup winding up anything left undone with C++17 standard
August 16 - August 23	**Final Evaluation**
After August 23	Completing secondary goals and contributing to other projects in casbin

Contributions prior to GSoC

PR #89, #90, #93, #94- feat: Added SyncedEnforcer

SyncedEnforcer was originally implemented in casbin-core which is written in GoLang. The implementation for this was straightforward in CPP except for ticker and concurrent functional callbacks per tick.

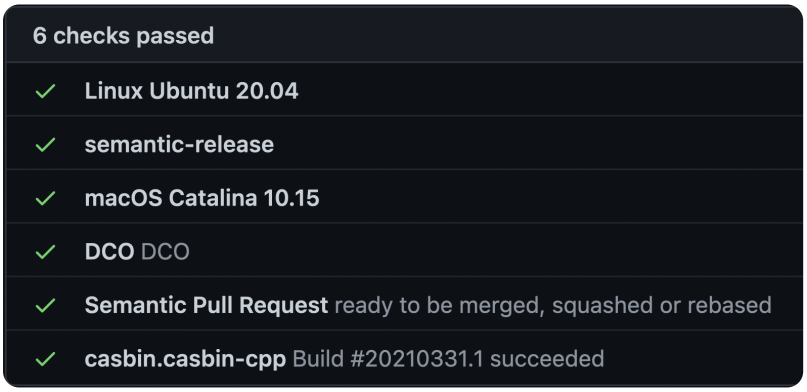
I created a custom concurrent ticker that takes in a function for callback asynchronously. The callback included a function to increment the counter for a number of updates and a call to LoadPolicy(). Apart from this, RAII-based mutex locks were also highlighted in this PR.

This implementation had an overhead of collecting void futures inside the ticker object at every tick.

PR #92 - feat: Replaced Travis-CI with GitHub Actions

During one of my PR's CI run, Travis did not trigger correctly which caused false negative or pending tests.

In this PR, I configured a GitHub Action which was similar to that of previous Travis configs for enchanced reliability and longevity.



PR #82 - feat: Replaced pragma with include guards

Building casbin-cpp on UNIX/Linux resulted in many unwanted warnings due to pragma guards.

We agreed upon going for traditional include gaurds as this removed a lot of clutter from build console on non Windows OS.

Why me for the project?

I've been fascinated by block-chain technology and the idea of decentralisation always grew upon me to learn more about it. Authorization and cryptography are the foundation for technologies like block-chain. The projects under Casbin seem to me like a perfect way to get a hands-on look into various types of authorization and get to see its real world applications. I've been programming for a while and have acquired some decent skills including the concepts of C++, OOPs, CI/CD, build systems and Software Development.

I've only recently started to study about Python and GoLang. The best way to learn these languages would be to make actual projects which I did and learned a lot from. I am confident enough that by the time I start working on the projects, I'd have studied about Python and GoLang well enough to get going. I make atomic commits with short and clean commit messages with well documented and well structured Pull Requests as I've demonstrated in my previous contributions. For all the reasons stated here, I think Casbin and its community is perfect for me to explore more into the world of authorization and open-source.

I am willing to devote around **40 hours per week** for this project. I've carefully looked upon the timeline and believe that it is something I'd be able to achieve this summer without rushing things up or alloting too much of time for trivial tasks. I would also love to continue contributing to Casbin through the secondary goals after GSoC period.