IoT Security Verification

FINAL REPORT

sdmay21-41

Megan Ryan
Kristin Rozier

Joshua French
Vincent Johnson
Jordan McKillip
Marcus Reecy

sdmay21-41@iastate.edu
https://sdmay21-41.sd.ece.iastate.edu/team.html

Revised: 4-25-2021/0.02

Executive Summary

Development Standards & Practices Used

Software: SV-COMP, Ubuntu 2004, ABET

Waterfall methodology

Summary of Requirements

Ubuntu 2004 VM machine/x86_64-linux to run and test the programs. A memory limit of 15 GB (14.6 GiB) of RAM, a runtime limit of 15 min of CPU time, and a limit to 8 processing units of a CPU conforming to standard SV-COMP guidelines.

Applicable Courses from Iowa State University Curriculum

SE/CPrE 185, COMS227, COMS228, COMS311, SE339, SE329

New Skills/Knowledge acquired that was not taught in courses

Through this project, we learned about some potential security vulnerabilities, SV-COMP, Linear Temporal Logic formulas, and Git management.

Table of Contents

List of figures/tables/symbols/definitions

Library - A set of prebuilt code that allows a developer to call or use some functionality of it. The developer chooses when and where to use the library.

Framework - A set of prebuilt code that allows a developer to add some functionality in preset locations. A framework controls the flow of code and a developer may edit some portions of the framework.

Verification Run - a non-interactive execution of a candidate on a single verification task, in order to check if the following statement is correct: *"The program satisfies the specification."*

Linear Temporal Logic Formula - a modal **temporal logic** with modalities referring to time. In LTL, one can encode **formulae** about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc.

Verification Task - consists of a C program and a specification, security property.

Property - a specification to be verified for a program.

IoT Code - any code that can be used to build an IoT device or has functionality that may assist in building an IoT device.

Benchmark- any code that has been tested for the different security validation properties. Once finished, the details will be mounted within a matrix repository.

1 Introduction

1.1 Acknowledgement

Our project's purpose is to expand BenchExec and SV-COMP and utilize the repository and tools provided by the non-profit organization ETAPS. We would also like to thank Muhamed Stilic for the work originally completed within developing the benchmarks in C.

1.2 Problem and Project Statement

The Internet of Things (IoT) is becoming more and more a part of people's everyday lives. Devices such as locks, cameras, and smart-speakers are just a very small view of all the ways our lives are going online. With all of these devices having important roles, being located in private places, and gathering loads of information, the security of them is much more prevalent as it would be problematic if it got into the wrong hands.

There are already some ways that the security of the code behind these IoT devices is being tested. However, there are a lot of security properties that aren't being as thoroughly checked. One of the ways is through a program called BenchExec. BenchExec is a security validation software that helps compare different software verification tools to help find which tools will suitably satisfy your needs. These tools test a variety of common security failings to ensure that the software can be validated and secure for any developing needs regarding the compatible c software. Our project is developing on BenchExec and expanding it to focus on IoT device code and test different IoT libraries.

The final goal of the project is to create a set of IoT benchmarks, and utilize the top three tools and security properties to verify said benchmarks. This will allow future competitions to validate future tools for IoT code.

1.3 Operational Environment

Ubuntu 2004 VM machine/x86_64-linux, a memory limit of 15 GB (14.6 GiB) of RAM, a runtime limit of 15 min of CPU time, and a limit to 8 processing units of a CPU.

1.4 Requirements

The software can be downloaded, it can be replicated, and evaluated.

The software should not require any special software on the competition machines; all necessary libraries and external tools should be contained in the archive.

The software reports its version within the readme and other documentation.

Remains free of unnecessary data, with only the core code and descriptions within the code, free of things such as test files.

The software is written and tested within the C programming language.

1.5 Intended Users and Uses

The intended users of this software will be developers and academia of IoT.

End users will be able to use and create their own developed benchmarks following the detailed analysis that has been implemented within the repository.

1.6 Assumptions and Limitations

Assumptions:

- Only using C IoT code and benchmarks

- BenchExec will run smoothly on a different Ubuntu versions
- Verification tasks will run correctly on BenchExec no matter who downloaded the file

Limitations:

- The monetary cost to produce the end result shall be zero
- VMs are limited to one core on 8 GB of RAM running Ubuntu
- We are limited to the end of spring semester to finish the project

1.7 Expected End Product and Deliverables

List of several well rounded IoT libraries and IoT Projects

> These IoT libraries will all come from open source code on the internet. A well rounded library will be one that does not have a profusion of dependencies in it. Some of these libraries may be slightly modified to make them into well rounded libraries. This list will then be run through IoT validation tasks to see how secure they are.

A running instance of BenchExec

> This instance of BenchExec will be set up on VMs provided by the university. The instance of BenchExec will be able to run the IoT verification tasks on the open source IoT libraries. The code will be modified and stored the majority of the time being in a private git repository.

IoT Benchmarks with Expected Results within designated tools and security properties.

> This set of verification tasks will be focused around the security aspect of IoT devices. Part of these may come from the BenchExec benchmarks repository, whereas others will have to be written on our own. These tasks will cover a slew of security issues with IoT devices, and will not focus on any one particular aspect/weakness.

2 Project Plan

2.1 Task Decomposition

Below is a list of our planned tasks, and some of the steps required within each task to complete it. Some tasks require specific steps, whereas others require more open-ended research and data collection. Most tasks build upon the knowledge found in the first task of general research.

- General Research
  - Team introductions
  - Vulnerabilities
  - SV-COMP
  - IoT code
  - Code libraries versus frameworks
  - Code verification tools
- Identify Milestones
  - Discuss with advisor and client
  - Have a clear end-goal decided on

    - Verify milestones with advisor and client
- Identify IoT Libraries for use - the team will be testing many libraries throughout the project
    - Create library benchmarks - based on data found in general research
    - Create IoT code benchmarks - based on data found in general research
    - Research available IoT libraries based on determined benchmarks
    - Choose a select amount of libraries for use
- Identify verification properties to test
    - Research which properties are tested often - knowledge of vulnerabilities from general research used
    - Choose properties not tested as often - partially based on IoT libraries decided upon
- Set up SV-COMP
    - Get access to an ISU virtual machine (each team member)
    - Decide on BenchExec tools to use for C - based on knowledge from general research
    - Set up BenchExec on virtual machines - uses knowledge from general research
- Design C Verification Tasks
    - Use knowledge from created Java verification tasks
- Run Verification Tasks
    - Utilize BenchExec
    - Plug in IoT libraries previously decided upon
    - Verify libraries based on previously chosen security properties
- Build a Series of Benchmarks and design Documentation
    - Using validated IoT Code from our tested benchmarks as well as the recorded information from a git repository.

2.2 Risks And Risk Management/Mitigation

For our project we have identified the following risks and risk mitigation plans:

Licensing Issues - a few IoT platforms require capital to use. Our mitigation strategy for this is to use open source IoT code and libraries.

Inadequate Design - a risk associated with a misunderstanding of the project's goals. Our mitigation strategy is to define our project problem and our statement and use those definitions to expand on our design of our project.

Team Dynamics - a risk with any project that has a team. Our mitigation strategy is to have weekly team meetings, address issues as they arise, and be proactive. In addition, several methods of communication have been set up to ensure that other team members are held accountable.

Developing wrong software functions - this risk can come from miscommunication or misunderstanding of the project requirements. A mitigation strategy for this risk is to have code peer reviewed and have weekly team meetings to go over functions that are required for our task.

Gold Plating - adding more features to a product that the client did not ask for. Our mitigation strategy for this is to keep within our project plan and use our weekly client meetings to stay within the scope.

Incompatible Libraries - we may run into a library that requires too much time to 'round' off. One way to mitigate this is to look for standard libraries in addition to checking libraries versus library definition.

Incompatible IoT Code - code that is heavily library dependent requires too much time to properly 'round' off to run as a validation task. Our mitigation strategy is to look for good IoT code commonly used in IoT and use that as a comparison tool with other IoT code to validate.

Time Constraints -as we run these verification runs our system will have to do model checking. This takes real time and since we have limited ram it may take a full day or more to run a verification run. Our mitigation strategy for this is to set soft limits for verification runs. For example, a normal run would be considered <24hrs, but anything >24hrs we will consider as an unknown failure.

2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

There are currently seven milestones for this project, and each of them fall under a specific deliverable:

The first milestone falls within the deliverable, "Project Research", and is described as being met once we have sufficiently researched key materials related to our project abstract. The metric is to have all team members aware and understanding the required information needed to advance to the next milestone.

The second milestone is met once the team has identified both the milestones and timelines of the project. This falls under the "Prepare Project Plan" deliverable. The metric will be designated by a proper timeline given and agreed upon by the project team so that other milestones can be properly accomplished within the required timeframe.

The third milestone, within the "Set-up BenchExec" deliverable, is met once the team builds the SV-COMP environment, and is able to run verification tasks composed of an IoT library and a chosen security property. The metric is when more than half of team members have created a sustainable environment and successfully ran BenchExec within their virtual workspace. Once that has been confirmed by other members of the project team will they proceed to the next milestone.

The fourth milestone is achieved once the team has successfully completed designs for benchmarks that can be tested with the given security tasks. The metric is to create test cases for both secure and unsecure code in order to test validation properties of the chosen security tasks.

Finally, the fifth milestone will be completed once we compile all of the benchmarks made from the previous milestone and allocate all the data and the process of making into a readable and functionable library. The metric will be the creation of the matrix where each of the verification tasks are seen and documented within a matrix or a git document respectfully.

2.4 Project Timeline/Schedule

The schedule for the overall project was as follows. It allowed for additional time to be taken within each of the different sections should delays be prevalent throughout. These long periods of time will allow us to break them down into smaller tasks. This way the project end date will not be delayed.

Introduce Team and Project (8/24/2020 - 8/31-2020) - In this section we introduced ourselves to our team, client, and advisor. We also were given a summary of what our project would consist of. As a team, we decided when meetings would be and set internal roles and responsibilities.

General Research (8/31/2020 - 9/8/2020) - We were assigned research topics by our advisor to understand our project in a further manner. We created slides containing our research and additional questions we had. These questions turned into further research.

Identify Milestones (8/31/2020 - 9/8/2020) - After being given an overview of the project we brainstormed general milestones that our team could use to guide our project. Though we came up with a set of milestones, we all agreed that this would continue when new goals arose.

Identify IoT Libraries for use (9/8/2020 - 4/15/2021) - One major portion of our project is to identify IoT libraries that we may use for verification tasks. This milestone started by identifying criteria that we may consider before picking a library. It continues to 4/15/2020 because we will continue to pick and choose new libraries as we continually design verification tasks.

Identify verification properties to test (9/8/2020 - 4/15/2021) - Verification properties are the properties that we test in a library to determine if it meets certain criteria (in our case, security). Verification properties will continually be edited as we consider new libraries. Because we are also considering new libraries until 4/15/2021, we must also identify new verification properties until this same time.

Get / Set up SV-COMP (9/22/2020 - 10/13/2020) - Our team must request and set up virtual machines such that SV-COMP and other testing tools may run. This time period is 3 weeks because we must wait on the ISU IT department to give us access to these VMs.

Design Verification Tasks for C(10/13/2020 - 4/15/2021) - Our team must use the chosen libraries to design verification tasks to run in SV-COMP. These verification tasks will be written on libraries specifically in C. As we pick new libraries we will also need to write new verification tasks for these libraries. Since we will be considering libraries until 4/15/2021, we must also write new verification tasks until this point.

Run Verification Tasks (11/24/2020 - 5/1/2021) - As our team writes verification tasks we will continually be running and testing them to make sure of completion. This process has a potential to take a long time and thus takes up most of our allotted schedule. After 4/15/2021 (When no new libraries will be chosen by our team), we will continually develop with the libraries that we have in our hands at that time. By 5/1/2021 we wish to have all verification tasks for the libraries in a complete state.

Build a Validated IoT Benchmarks with Validated Property (11/24/2020 - 5/1/2021) - Validated IoT Code from our benchmarks will then be compiled into a validated IoT library for future users to utilize.

2.5 Project Tracking Procedures

The group has decided to utilize a waterfall method and a gantt chart to track the progress of the project in its various stages. The Waterfall model will be used to make sure that all members are aware of what parts of the project are being worked on at what time. Should additional information force the group to focus on one specific project within the project, additional time will be allocated within the waterfall model so that everything is kept on track for our deliverable due date.

Messages between members will be communicated through Slack. Any vocal calls will be done through either Discord or Zoom at designated meeting times. Code and all progress can be evaluated and checked within a shared Git repository.

2.6 Personnel Effort Requirements

Task	Personal Hours
General Research	60-100
Identify IoT Libraries for Use	60-100
Identify verification properties to test	60-100
Get/Set up SV-Comp and related tools	60-100
Design Verification Task for C	100+
Run Verification Tasks and input data	100+

2.7 Other Resource Requirements

We will be using virtual machines provided by the university to use BenchExec to build and run our verification tasks.

2.8 Financial Requirements

All code and virtual machines are provided via the university, and all code is personally made or publically available. As such, currently, the project requires no financial support in order to proceed.

3 Design

3.1 Previous Work And Literature

CWE [1]
CWE is a website that contains all of the most common software weaknesses and vulnerabilities. Each year the list is updated with a new set of common weaknesses. Our project will involve taking some of these weaknesses and determining if a new security property should be developed.

SV-COMP [2]
SV-COMP is a competition whose goal is to publicly efficiently test and verify software. SV-COMP works to create and maintain a set of programs and security properties. These properties are made publicly available for researchers and developers. Our project revolves around using these properties (and making new ones) in order to build a list of security tested libraries.

Amnesia 33 [3]
Amnesia 33 is a comprehensive list of vulnerabilities that are prevalent in several pieces of well known code. It provides the nature of the vulnerabilities, as well as linked resources to the CWE to learn more about the vulnerabilities and the code that has them.

3.2 Design Thinking

There are a few aspects that shaped our design:

Find libraries and/or code to implement into verification tasks. Our team needs to put together a list of IoT libraries that we are able to run verification tasks on. A library is defined as a set of prebuilt code that allows a developer to call or use some functionality of. A library is different from a framework. A framework is defined as prebuilt code that controls the flow of the program. A library differs from this such that a developer can use it whenever they choose. Iot code is defined as any code that can be used to build an IoT device or has functionality that may assist in building an IoT application.

After considering these design aspects we were able to make a few more design choices that came up during the 'ideate' phase:

Find IoT related security properties. Our team needs to expand on current properties listed in SV-COMP for C that relate to both security and IoT.

Create Verification tasks. Our team needs to create and expand on current verification tasks listed in SV-COMP. We may use the following definition for a verification task: A set of C or Java programs and a specification, a security property.

What security properties will we cover? There are many security properties that are already listed in SV-COMP. Our team will look at the Common Weakness Enumeration (CWE) website and compare the weaknesses listed there that might relate to security. In the end, we will have a comprehensive list of properties that encompass SV-COMP and CWE.

What security properties will we create? As part of our deliverables, the creation and implementation of a new, unique property was to be tested alongside the chosen security properties.

3.3 Proposed Design

Through our research in the SV-COMP repository and its property files used in validation runs for BenchExec, creating benchmarks to be tested and the documentation to create them. We have discovered that we can use BenchExec and its tools provided to test properties that align with our chosen security standards.

Our proposed design is as follows:

BenchExec is mainly used to validate tools against predefined programs and predefined properties and gives each tool a score. With the tools score information for a particular property, we can leverage BenchExec and the chosen tool with the highest score for a property to validate IoT Code.

C IoT Benchmarks:
We can build a validation task, and benchmarks, within BenchExec which consists of a property and a program. The property of the validation task is a defined Linear Temporal Formula. Whereas the program is the code that will be validated against the property. BenchExec already has pre-defined properties for C those of which, valid-memsafety, valid-memcleanup, no-overflow we chose to be security related. Once a validation task has been compiled the BenchExec, a runnable, will then execute the validation task in a validation run. The validation run will then give a result of "True", "False", or "Unknown".

Through these tests, we were able to meet the following requirements:

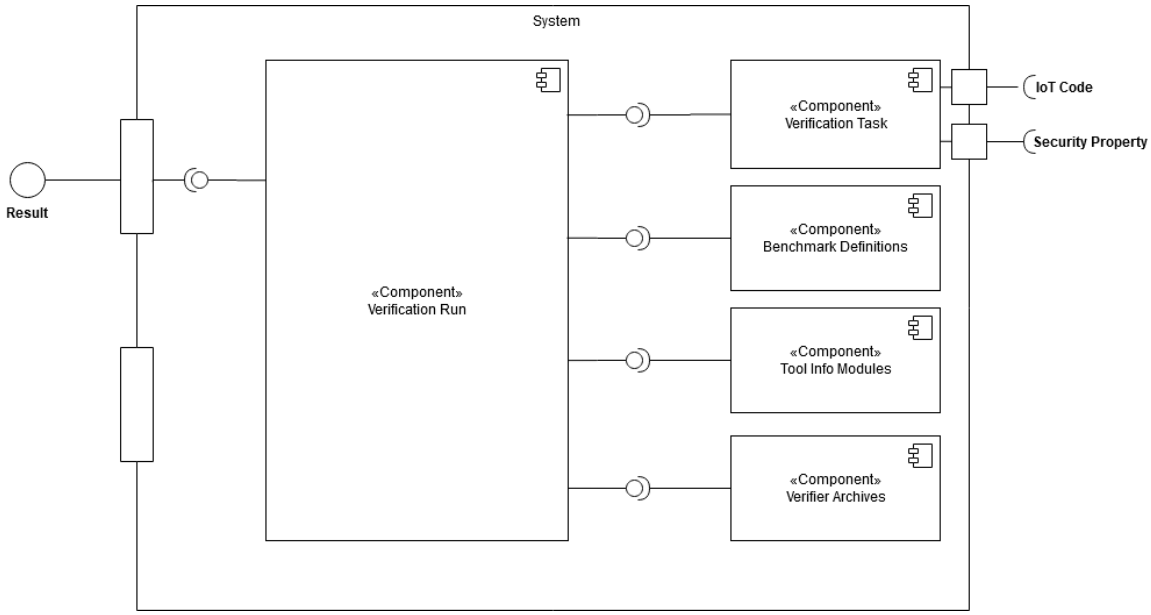
- Report validation result of the security property (Functional)
- Report its version (Functional)
- Can be downloaded, replicated and validated (Functional)
- File Package contained all libraries and tools (Functional)
- File Package contained no unnecessary data, code, etc (Functional)
- Was written in C (Functional)

In addition, our proposed design will need to meet the following IEEE standards:

- IEEE 802.11ai-2016 - initial setup methods and their security. Implemented because the setup of original methods requires direct standardization that must be followed if we wish for others to use our proposed verification task.
- IEEE 1012-2016 - verification and validation of systems, software, and hardware life cycles Implemented because the scope of the project and the deliverables must conform and satisfy the intended use and user requirements.
- IEEE P2933 - creating a framework for IoT data and device interoperability in the clinical region that incorporates the values of TIPPSS. Implemented because the project must hold true and be secure for anything related to protecting the confidentiality of any device validation or interoperability of the internet of things when connected to hardware systems which does fall under our project deliverables.

These standards will need to be met when we are modifying IoT Code and/or selecting IoT libraries to test.

Furthermore, based on the above Design Plan we designed the following UML diagram for our design.



3.4 Technology/Security Considerations

IoT devices are currently being produced at a very significant rate due to the high demand of developing the so-called "smart" houses and lifestyle. This results in fast production of devices without considering the quality of security incorporated. Our project can help validate some security properties without slowing down the development process too much and thus be applicable to a wide range of users and a potential large amount of code.

When it comes to the security, each of the codes are being tested to ensure that they can be implemented in a way that removes their security vulnerabilities. Should the project be successful and the code implemented, the security of the code will be validated.

3.5 Design Analysis

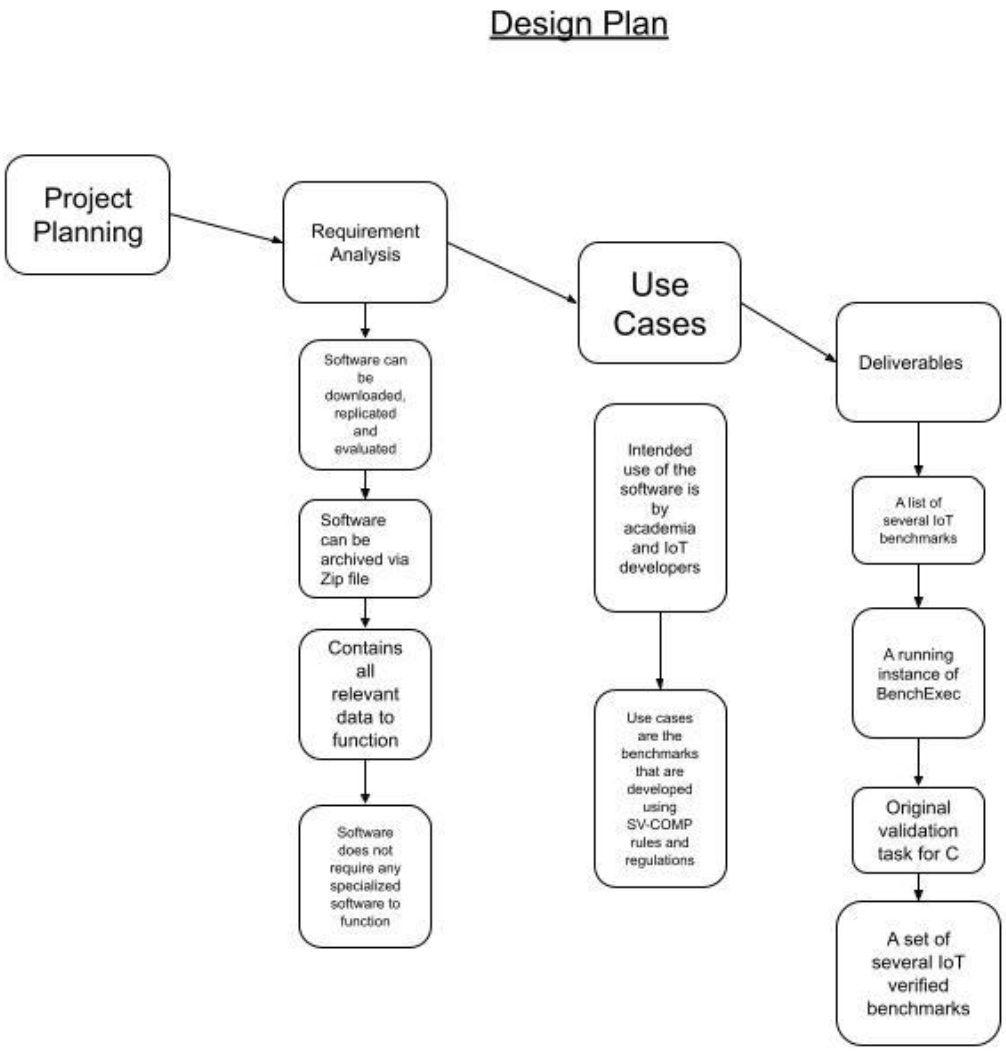
This design has been proactively tested. The IoT code was successfully implemented into a verification task and had a successful verification run on the specified machines, creating an instance to be tested like the other benchmarks. Through the use of constant revisions within the code, it was able to be imported into the VM's and tested against the use cases.

3.6 Development Process

Our team will be using the waterfall model for our development process for this project. This is largely due to the fact that we had to slowly learn/alter our project goals as we move forward. It is important for us to fully check in with our client at each step of the way so as to ensure that we are on the right track. Through the use of systems such as Slack, Discord, and Git, we are able to keep each other informed of any developments. Any advancements with code are stored within a branch with the git repository, and after it is verified, merged within the main branch. All work is documented for others to use, and progress for each team member is updated during designated meeting times using Discord or Zoom.

3.7 Design Plan

Our Design Plan follows four overall steps which can each contain substeps. The four main steps are as follows: project planning, analysing requirements, identifying use cases, and producing deliverables. Three out of the four main steps are made up of varying amounts of substeps. These steps and substeps are outlined in the diagram found below.

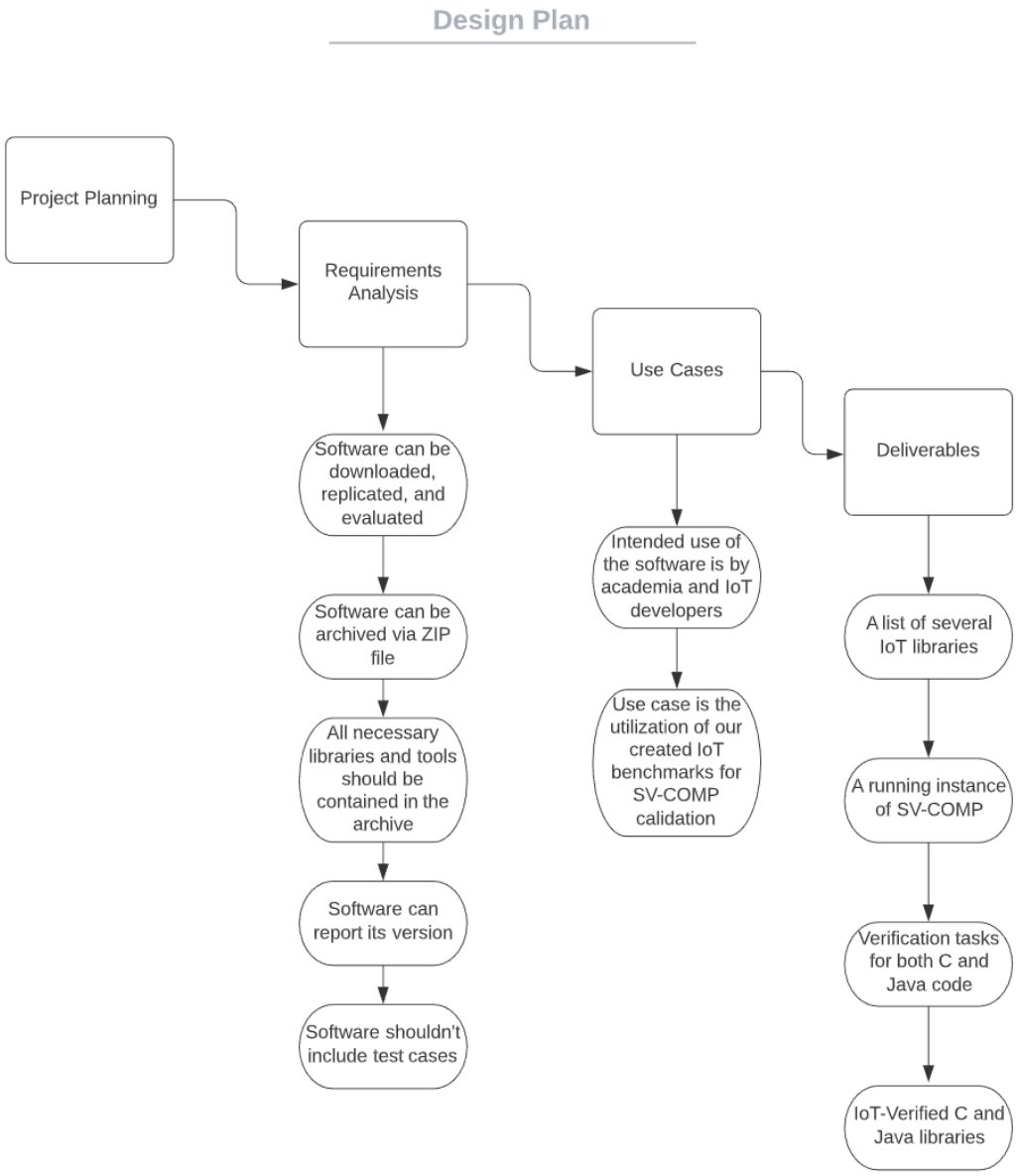


3.8 Evolution from original Project Plan

The original project plan involved testing and implementing properties that, through miscommunication of products with our supervisor, lead us to creating several validation properties that were unneeded. From there, the design property to develop the benchmarks and utilize the

different tools to test the allocated security properties became the forefront. Because of this, many problems within the different implementations were scrapped, and the design properties to create our own IoT properties were not met. Instead, the benchmarks for the deliverables were the focus and created, in addition to the creation of the steps that lead to the Operation manual located in Appendix [1].

Originally, the design had the following diagram to explain it the project steps:



4 Testing

4.1 Unit Testing

Once we have created our benchmarks, each of the security property tasks for C will need to be validated using verification runs. These properties are as follows: No Overflows, Memory Safety, Reach Safety, and Memory Cleanup. For each of the properties, we will need unit tests for C to make sure each IoT benchmark successfully runs according to its original design. Some testing will be required to make sure that the environment is set up correctly. Since we also will have to round out some of the IoT projects that we pull, testing will be conducted to show that the original functionality of the libraries remains unchanged.

4.2 Interface Testing

BenchExec is the main interface that is used in our project. It is an interface that is not developed by us, but is simply being used to run simulations. Being that it was not developed by us, we will not be testing it. BenchExec will be the program that is used to test our tools as well as our IoT code.

4.3 Acceptance Testing

In order to demonstrate that the design requirements are being met, we will conduct black-box testing. This way we can show through the user perspective if there are any discrepancies based on the specifications. It also helps with having an objective perspective and avoids developer bias. By conducting acceptance testing through the black box method, it will be in terms that are quickly understood by everyone, including the client.

4.4 Results

After extensive testing, we have compiled a matrix with a total of 31 benchmarks that have been tested with the given properties. These benchmarks include patched and vulnerable states of code in order to ensure that the different properties and tools that were being tested were working as expected. The following is the result of the tool testing portion of our research:

IoT BENCHMARKS	CPA-SEQ				PredatorHP				VeriAbs**			
	No Overflows	Mem Safety	Reach Safety	Mem Cleanup	No Overflows	Mem Safety	Reach Safety	Mem Cleanup	No Overflows	Mem Safety	Reach Safety	Mem Cleanup
Azure Plug and Play Bridge_patched.c	T	T		T	T	T		?	T	T		T
Azure Plug and Play Bridge_vulnerable.c	T	T	T	T	T	T	?	?	T	T	T	T
LightBlueLatch_Sensor_patched.c	T	?	T	?	U	U		?	T	T	T	T
LightBlueLatch_Sensor_vulnerable.c	T	T	T	T	U	U	?	?	T	T	T	T
Particle PI Camera_patched.c	T	U		?	U	U		?	T	T		T
Particle PI Camera_vulnerable.c	T	U	T	?	U	U	?	U	T	T	T	T
mqtt_num_rem_len_bytes_patched.c	*T*	*T*	*T*	*T*	T	T	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_num_rem_len_bytes_vulnerable.c	*T*	*T*	*T*	*T*	U	F	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_msg_id_patched.c	*T*	*F*	*T*	*T*	T	T	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_msg_id_vulnerable.c	*T*	*F*	*T*	*T*	T	T	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_pub_msg_ptr_patched.c	*T*	*F*	*T*	*?*	U	U	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_pub_msg_ptr_vulnerable.c	*T*	*F*	*T*	*?*	U	F	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_pub_topic_patched.c	*T*	*F*	*T*	*T*	T	T	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_pub_topic_vulnerable.c	*U*	*F*	*T*	*T*	U	U	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_pub_topic_ptr_patched.c	*T*	*F*	*T*	*T*	T	T	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_pub_topic_ptr_vulnerable.c	*T*	*F*	*T*	*T*	U	F	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_publish_msg_patched.c	*U*	*F*	*T*	*?*	U	U	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_publish_msg_vulnerable.c	*U*	*F*	*T*	*?*	U	U	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_rem_len_patched.c	*?*	*?*	*?*	*T*	U	U	*?*	*U*	*T*	*T*	*T*	*T*
mqtt_parse_rem_len_vulnerable.c	*T*	*F*	*T*	*T*	U	F	*?*	*U*	*T*	*T*	*T*	*T*
io_dig_patched.c	*T*	*T*	*F*	*T*	*T*	*T*	*U*	*T*	*?*	*?*	*U*	*U*
io_dig_vulnerable.c	T	T	F	T	F	F	U	T	?	?	U	U
clear_tempature_data_patched.c	T	T	T	T	T	T	U	T	T	T	T	T
clear_tempature_data_vulnerable.c	T	F	T	T	U	F	U	U	T	T	T	T
temptoa_patched.c	T	T	T	T	T	T	U	T	T	T	T	T
temptoa_vulnerable.c	T	T	T	T	T	T	U	T	T	T	T	T
count_lw_bus_error_patched.c	T	T	T	T	U	U	U	U	T	T	T	T
count_lw_bus_error_vulnerable.c	T	F	T	U	U	F	U	U	T	T	T	T
read_temperature_data_patched.c	U	F	T	U	T	T	U	T	T	T	T	T
read_temperature_data_vulnerable.c	U	F	T	U	U	U	U	U	T	T	T	T
select_lw_bus_patched.c	T	T	T	T	T	T	U	T	T	T	T	T
select_lw_bus_vulnerable.c	T	F	T	T	U	F	U	U	T	T	T	T
search_sensors_patched.c	U	T	T	T	U	U	U	U	T	T	T	T
search_sensors_vulnerable.c	U	T	T	T	U	U	U	U	T	T	T	T
fnet_ip6_ext_header_handler_result_t _fnet_ip6_ext_header_handler_options_patched	T	IF	U	U	U	IF	U	U	T	T	T	T
fnet_ip6_ext_header_handler_result_t _fnet_ip6_ext_header_handler_options_vulnerab	T	F	U	U	U	F	U	U	T	IT	IT	T

For each of the different tests, the results could be Correct True/False (T/F), or Incorrect True/False (IT/IF), Unknown (U) or Tool Error(?). For each of the different benchmarks, three tools were tested in order to ensure that a wide range of coverage and testing could occur.

For CPA-SEQ, the tool was chosen as it is the best of the open source tools that can test No Overflow. This property ensures that the benchmark can be tested so that the code doesn't allow for an overflow error to occur, creating a vulnerability where the code can be written into without administrative access.

PredatorHP was chosen as it was one of the best for memory safety. It looks at the code within the benchmarks that could lead to buffer overflows and dangling pointers as C allows for pointers that can be implemented as direct memory access. They don't have any bound checking natively unlike java, so PredatorHP checks the code to make sure that the memory is safe within the bounds of the code.

VeriAbs was chosen to check Reach Safety within the benchmarks. It takes the code and checks if any pieces of the code have arrays that are unreachable and therefore vulnerable. In addition, it checks the variability of bit vectors, the control flow and the integer vectors as well as checking the floating point arithmetics that could lead to exploitation within the benchmark among other properties.

With these tools, we designed our benchmarks like the following code:

```
extern uint8_t __VERIFIER_nondet_uint8_t(void);
extern uint16_t __VERIFIER_nondet_uint16_t(void);
extern int16_t __VERIFIER_nondet_int16_t(void);
extern uint32_t __VERIFIER_nondet_uint32_t(void);
extern int32_t __VERIFIER_nondet_int32_t(void);
extern float __VERIFIER_nondet_float(void);

uint8_t mqtt_num_rem_len_bytes(const uint8_t* buf);

int main() {
        while (1) {
                const uint8_t* buf = (void*) __VERIFIER_nondet_uint8_t();
                mqtt_num_rem_len_bytes(buf);
        }
}

uint8_t mqtt_num_rem_len_bytes(const uint8_t* buf) {
        uint8_t num_bytes = 1;

        //printf("mqtt_num_rem_len_bytes\n");

        if ((buf[1] & 0x80) == 0x80) {
                num_bytes++;
                if ((buf[2] & 0x80) == 0x80) {
                        num_bytes ++;
                        if ((buf[3] & 0x80) == 0x80) {
                                num_bytes ++;
                        }
                }
        }
        return num_bytes;
}
```

It takes all of the different parts of the code, along with a main function that simulates the running

of the code so that the tools can look through each of the code's moving parts in order to succeed.

It leads to the showing of a result like the two below:

```
executing run set 'sdmay21-41_no-overflow.ReachSafety-no-overflow'     (3 files)
03:12:19   LightBlueLatch_sensor/LightBlueLatch_Sandbox_Keycode.yml    true                    1.05    0.79
03:12:20   Particle_Pi_Camera/Particle_Pi_Camera.yml                   unknown                 0.58    0.54
03:12:21   Azure_Plug_Play_Bridge/SerialPnP.yml                        TIMEOUT               900.51  450.30

executing run set 'sdmay21-41_reach-safety.ReachSafety-unreach-call'     (2 files)
03:19:53   LightBlueLatch_sensor/LightBlueLatch_Sandbox_Keycode.yml    UNKNOWN                 0.08    0.29
03:19:53   Azure_Plug_Play_Bridge/SerialPnP.yml                        UNKNOWN                 0.07    0.29

executing run set 'sdmay21-41_mem-safety.ReachSafety-memsafety'     (3 files)
03:19:54   LightBlueLatch_sensor/LightBlueLatch_Sandbox_Keycode.yml    true                    0.87    0.29
03:19:55   Particle_Pi_Camera/Particle_Pi_Camera.yml                   unknown                 0.57    0.54
03:19:56   Azure_Plug_Play_Bridge/SerialPnP.yml                        TIMEOUT               900.50  450.41
```

The above figure shows our tool verifying a true for LightBlueLatch on no-overflow and memsafety. However the tool is unsuccessful on both Particle_PI_Camera and Azure_Plug_Play_Bridge. The yellow 'unknown' represents a tool failure whereas a purple 'unknown' represents an error with our benchmark.

```
executing run set 'sdmay21-41_no-overflow.ReachSafety-no-overflow'     (3 files)
20:26:07   LightBlueLatch_sensor/LightBlueLatch_Sandbox_Keycode.yml    true                   16.38    6.95
20:26:15   Particle_Pi_Camera/Particle_Pi_Camera.yml                   true                   20.84    8.34
20:26:23   Azure_Plug_Play_Bridge/SerialPnP.yml                        true                   18.48    7.60

executing run set 'sdmay21-41_reach-safety.ReachSafety-unreach-call'     (2 files)
20:26:31   LightBlueLatch_sensor/LightBlueLatch_Sandbox_Keycode.yml    true                   15.93    6.40
20:26:38   Azure_Plug_Play_Bridge/SerialPnP.yml                        true                   19.41    7.65

executing run set 'sdmay21-41_mem-safety.ReachSafety-memsafety'     (3 files)
20:26:46   LightBlueLatch_sensor/LightBlueLatch_Sandbox_Keycode.yml    true                   16.16    6.22
20:26:53   Particle_Pi_Camera/Particle_Pi_Camera.yml                   true                   21.19    8.32
20:27:02   Azure_Plug_Play_Bridge/SerialPnP.yml                        true                   19.04    7.58
```

The above figure shows how the tool verified the benchmarks when we fixed the tool error and benchmark errors.

5 Implementation

The implementation of this code and the resulting documentation is to be uploaded as passable benchmarks to the SV-Comp regulated competition. With that, our IoT benchmarks will be available for future competitions for software verification tools to be tested against. In addition, with our documentation future users will be able to create their own benchmarks for software verification.

6 Appendix [1]- Operation Manual

1. **Download and setup BenchExec**
    a. *Install Python 3.6 or newer*
        i. sudo apt-get install python3.x
    b. *Install BenchExec on Ubuntu*
        i. cd ~
        ii. sudo add-apt-repository ppa:sosy-lab/benchmarking
        iii. sudo apt install benchexec
        iv. adduser <USER> benchexec
        v. reboot system
    c. *Install pqos_wrapper*
        i. Create a group named msr
        ii. Build the project as an independent executable using pyinstaller
            1. pyinstaller --onfile bin/pqos_wrapper
        iii. Install the binary in a location such as /usr/local/bin/pqos_wrapper
        iv. Set permissions for all users to use pqos_wrapper:
            1. sudo chgrp msr /usr/local/pqos_wrapper
            2. sudo chmod a+x,g+s /usr/local/bin/pqos_wrapper
            3. sudo setcap cap_sys rawio=eip /usr/local/bin/pqos_wrapper
        v. Ensure that the group msr has read-write permissions to /dev/cpu/*/msr
            1. sudo chgrp msr /dev/cpu/*/msr; sudo chmod g+rw /dev/cpu/*/msr
    d. *Verify Benchexec Setup*
        i. python3 -m benchexec.check_cgroups

2. **Download SV-Comp Benchmarks**
    a. *cd ~*
    b. *mkdir SV-COMP*
    c. *cd SV-COMP/*
    d. *git clone https://github.com/sosy-lab/sv-benchmarks.git*

3. **Download SV-COMP Benchmark Definitions**
    a. *cd SV-COMP/*
    b. *git clone https://gitlab.com/sosy-lab/sv-comp/bench-defs.git*

4. **Download SV-COMP's Tool Repository**
    a. *cd ~*
    b. *git clone https://gitlab.com/sosy-lab/sv-comp/archives-2021.git*

5. **Implement the tools in your environment**
    a. *cd ~*
    b. *mv ~/archives-2021/<Tool>.zip ~/*
    c. *unzip ~/<Tool>.zip*

6. **Create your own C Benchmarks**
    a. *Import benchmarks (.c file) into ~/sv-comp/sv-benchmarks/c/<benchmark-dir>*
    b. *Import .yml file for each benchmark into ~/sv-comp/sv-benchmarks/c/<benchmark-dir> with the following contents:*

```
format_version: '2.0'
input files: '<filename>.i'

properties:

  - property_file: ../properties/<propertyfile>.prp
    expected_verdict: <True / False>

options:
    language: C
    data_model: ILP32
```

    c. *Create a README in <benchmark-dir>*
    d. *Create a Makefile in <benchmark-dir> with contents:*
    LEVEL := ../
    include $(LEVEL)/Makefile.config

    e. *Create/Edit .set file in base /c/ directory with contents*
    <benchmark-dir>/*.yml

    f. *Compile .c files with Benchexec exceptions*
        i. cd <benchmark-dir>/
        ii. make
    g. *Create .i files*
        i. cd <benchmark-dir>/
        ii. gcc -E <.c file> -o <i.file> -P -m64
    h. *Verify integrity of benchmarks*
        i. cd /c/
        ii. python3 check.py
        iii. If any errors check the above steps.

7. **Run the tool with Benchexec**
    a. *First check to see if you meet the tools requirements to run, if you do not modify tools xml file*
        i. *vi ~/SV-COMP/benchmark-defs/<tool_file>.xml*
        ii. *Edit any hardware specifications for your machine.*
    b. *cd ~/<Tool_File>/*
    c. *benchexec ../SV-COMP/benchmark-defs/<tool_file>.xml -r <benchmark definition>*

7 Appendix [2]- Alternative Implementation

When the project began, our research and our goal was to develop our own IoT library. Not simply benchmarks, a library. With that in mind, our entire design was predicated on being able to process and code this enormous undertaking. After repeated talks with our client, the idea was scrapped, and the new version that has been shown to you. The scrapping and the implementation of the new system caused no small amount of headaches and confusion within the group after losing two of our members.

Originally, the project was set to include Java IoT libraries. The original intention was to create a Java IoT library that could be copied and distributed for others, but the time limitations of the project didn't allow us to work with java, as well as our understanding of the original guidelines proved far too much.

8 Closing Material

8.1 Conclusion

After a long and difficult road, the project has allowed us to work on and grow in our knowledge of IoT and benchmark design. We hope that the documentation that we have collected as well as the knowledge that we have tested will be able to help others as they try to continue to validate their security properties in the ever changing world of the Internet of Things.

We'd like to thank both Doctor Rozy and Megan Ryan for assisting and guiding us through this process.

8.2 References

[1] Cwe.mitre.org. 2020. *CWE - Common Weakness Enumeration*. [online] Available at: <https://cwe.mitre.org/> [Accessed 25 October 2020].

[2] sv-comp.sosy-lab.org. 2020. *SV-COMP 2021*. [online] Available at: <https://sv-comp.sosy-lab.org/2021> [Accessed 25 October 2020]

[3] Amnesia 33 2020 *Forescout Active Defense for the Enterprise of Things* Available at: <https://www.forescout.com/company/resources/amnesia33-how-tcp-ip-stacks-breed-critical-vulnerabilities-in-iot-ot-and-it-devices/> [Accessed December 2020]