SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

# Study of Models for Runtime Remote Attestations

Submitted by

Jon Kartago LAMIDA

Thesis Advisor

Prof. JIANYING ZHOU and FLAVIO TOFFALINI

ISTD (MSSD)

A thesis submitted to the Singapore University of Technology and Design in fulfillment of the requirement for the degree of Master of Science in Security by Design, ISTD (MSSD)

July 27, 2021

# Declaration of Authorship

I, Jon Kartago LAMIDA, declare that this thesis titled, "Study of Models for Runtime Remote Attestations" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

*"By (the Token of) Time (through the ages), Verily Man is in loss, Except such as have Faith, and do righteous deeds, and (join together) in the mutual teaching of Truth, and of Patience and Constancy."*

The Quran - The Epoch

SUTD

# *Abstract*

ISTD (MSSD)

Master of Science in Security by Design

**Study of Models for**
**Runtime Remote Attestations**

by Jon Kartago LAMIDA

Runtime remote attestation enable attesting application to ensure there is no control flow attack that alter the intended behavior of the program. Prior to the ScaRR [29] , remote attestation was only feasible for embedded system and there was no scalable solution for complex programs. This thesis present the implementation of ScaRR offline measurement using LLVM and analyze the performance of the computation.

Flavio leave the abstract as last point ◁

# *Acknowledgements*

The acknowledgments and the people to thank go here, don't forget to include your project advisor. . .

# Contents

# List of Figures

# List of Tables

# List of Listings

*This thesis is dedicated to my family*

# Chapter 1

# Introduction

Memory corruption errors have been a big security problem for almost 40 years [28, 30]. The errors offtenly happen due to the use of unsafe languages, mainly C and C++ in building applications. This safety issue becomes a significant attack surfaces that affect the security of many critical applications. The adversaries can hijack the program and taking the control to achieve their goals. Such attack gives impact to wide range applicatinos from distributed systems running in the cloud to IoT devices that are ubiquitous nowadays.

Since the identification these vulnerabilities, a continues arm races have been ongoing between adversaries and defenders. As attackers find a vulnerability then a different defense mechanisms is invented. After that, attackers and researchers find another vulnerability that can circumvent or disable the defense. Or sometime, the solutions are not just practical to be deployed. Across the year we are having solution such as non-executable stack (NX) or Data Execution Prevention (DEP) $W \bigoplus R$ [30] Control Flow Integrity (CFG) [1], ASLR [15], Stack Canaries [4] and more. Unfortuantely the war is not over.

In 2016, Abera et al published a solution called C-Flat for detecting a control-flow attack in runtime [2]. The detection is performed by mechanism called remote attestation [14]. The research focused on attesting runtime control-flow attack on IoT or other embedded devices. Since then, many different runtime remote attestation schema are introduced [13, 31, 16, 12, 3, 17, 27]. Like C-Flat, most of those remote attestation scheme are targetting embedded systems. One unique runtime remote attestation schema name ScaRR tries to cover remote attestation beyond embedded application but also made it work for complex system [29].

This thesis study different model for runtime remote attestation. We zoom in the ScaRR implementation due to the unique strength on making the attestation work for complex system.

## 1.1 Motivation

In this thesis we want to answer these questions.

- What are different remote attestation model that is available now and how do they differ?
- How to implement offline program analysis for the ScaRR novel model for remote attestation?
- How is the performance of the model?

## 1.2 Related Works

In this section we briefly discuss on the different runtime remote attestation model. Specifically we discuss how different attestation scheme encode the offline program representations.

C-Flat [2] is the first remote attestation scheme to detect runtime control flow attack for embedded systems. Figure 1.1 shows how C-Flat generates offline measurement by traversing all possible path of program from start node to the termination node. In each node, C-Flat hashes the node ID and the hash of previous node. In the first node, since there is no previous hash, we pass 0. This creates hash chains which is stored as offline measurement database.

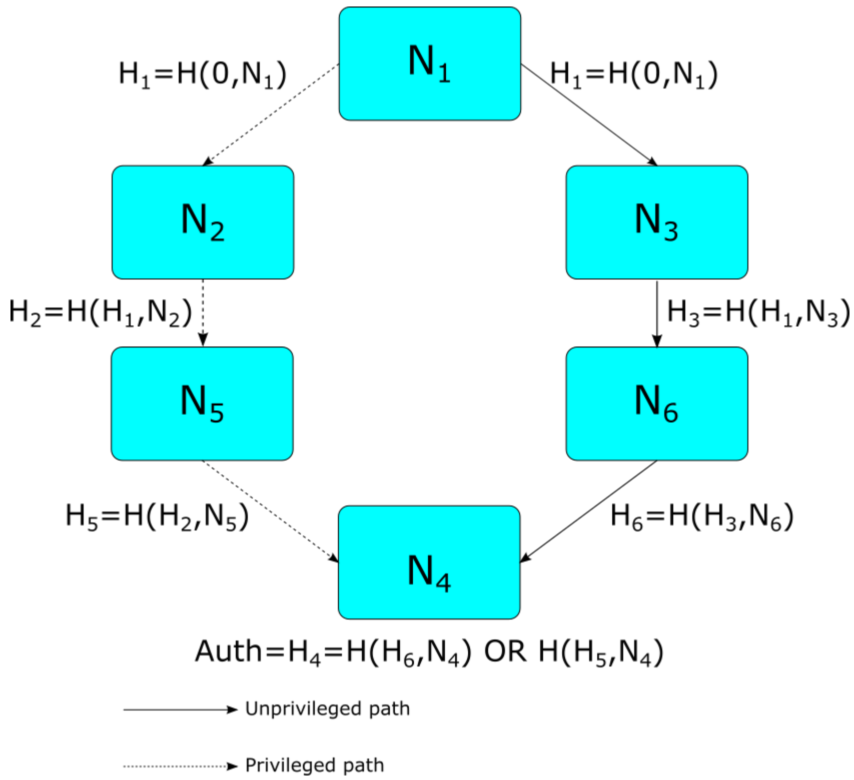


FIGURE 1.1: C-Flat

With that control-flow model, C-Flat can attest the exact control-flow path of the program. C-Flat also doesn't need source code because the offline measurement can be run on the binary program. However, C-Flat has one limitation —which is stated by the authors themselves— on its inefficiencies for having to explore all possible path from program control flow graph [2].

Lo-Fat[13] improves C-Flat by using hardware support for control flow attestation. See Figure 1.2 for the Lo-Fat architecture. The hardware will intercept the instructions and process it in the components called branch filter and loop monitor. With this hardware support, Lo-Fat incurs no performance overhead. However, Lo-Fat control-flow representation still inherits C-Flat approach, therefore it still induce high verification cost.

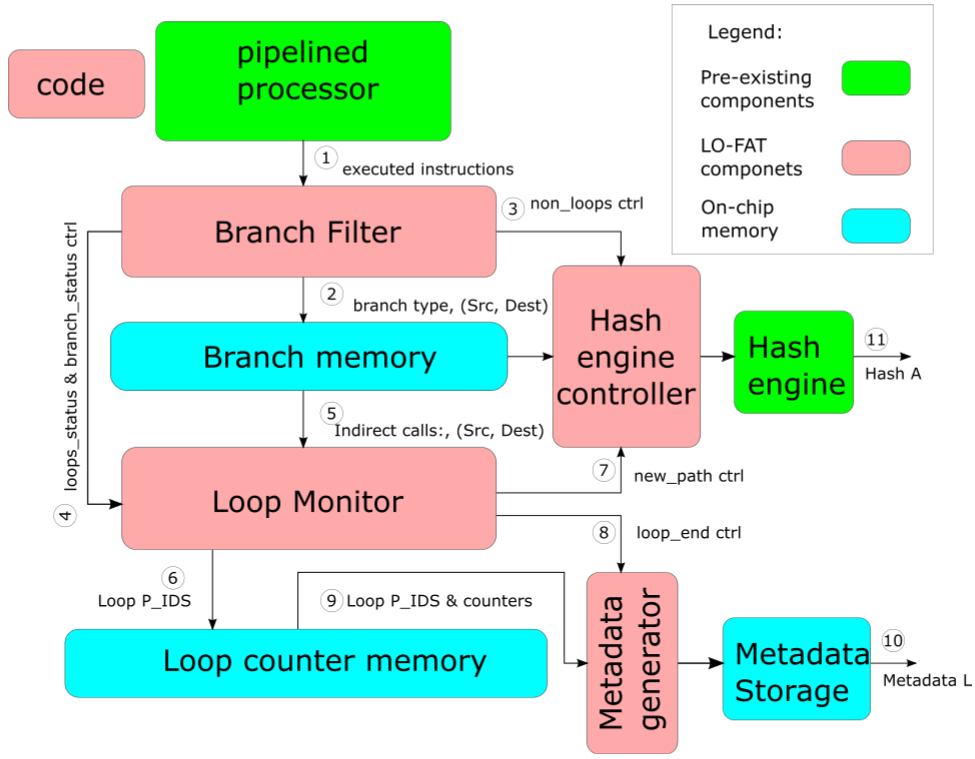


FIGURE 1.2: Lo-Fat

Atrium [31] is remote attestation scheme that can provide resiliency against physical memory attack where adversaries can exploit the property of Time of Check Time of Use (TOCTOU) during attestation. In this paper author are describing memory bank attack where adversary can control instruction fetches to benign memory area when attestation is running and direct the fetch to the malicious area otherwise.

The offline measurement are calculated slightly different compared with C-Flat and Lo-Fat. In Atrium, the verifier perform one-time pre-processing to generate CFG of the program and computes cryptographic hash measurement over the instructions and addresses of basic blocks. C-Flat are only hash the node ID. While this approach can mitigate the TOCTOU attack, the offline measurement generation still grow exponentially as the complexity of the program grow.

LiteHax [12] is hardware assisted remote attestation scheme that allow verifier to detect these different attacks:

- control-data attack such as code injection or code reuse attack like ROP
- non-control-data attack
- data-only attack such us DOP which do not affect control flow

Different with the previous remote attestation scheme, the offline measurement phase of LiteHax are only generates program CFG without calculating any hash over all control flow and data flow events. However, in the online prover-side verification time, prover are still computing hash and sending it as report to the verifier. Verifier

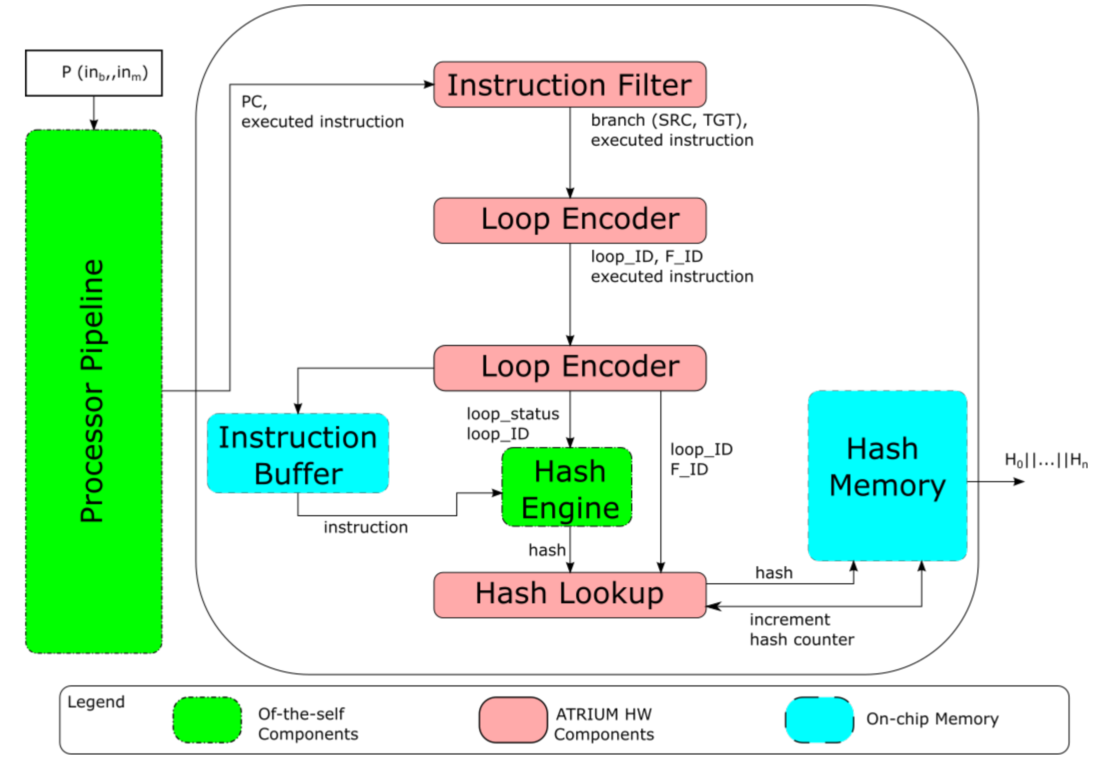


FIGURE 1.3: Atrium Architecture

runs symbolic execution and incremental forward data-flow analysis without doing any lookup to offline measurement database. LiteHAX architecture can be seen in figure 1.4.

Diat [3] is remote attestation scheme that can attest data integrity and control-flow of autonomous collaborative network systems. To improve efficiency of attestation, the program attested must be decomposed into small interacting modules. Data-flow monitoring is to be setup between critical modules. Control path attestation is being done against novel execution path representation using multiset has (MSH) function [9]. See the control flow monitor logic in figure 1.5. The use of MSH makes some execution order of the program cannot be reconstucted.

OAT [27] is remote attestation scheme to attest operation integrity of embedded device. OAT defines two type of measurements for control flow attestation: a trace (for recording branches and jumps) and a hash (for encoding returns). These two measurements are encoded as $H = Hash(H \bigoplus RetAddr)$ which called as attestation blob. Figure 1.6 shows the OAT control-flow attestation.

During verification, verifier reconstruct paths from the attestation blob. The control flow violation is identified when CFI check against an address is failed or mismatched between hash and trace.

Although OAT does not encounter the combinatorial hash explosion in C-Flat, there is a verification overhead since verifier needs to reconstruct the attestation blob.

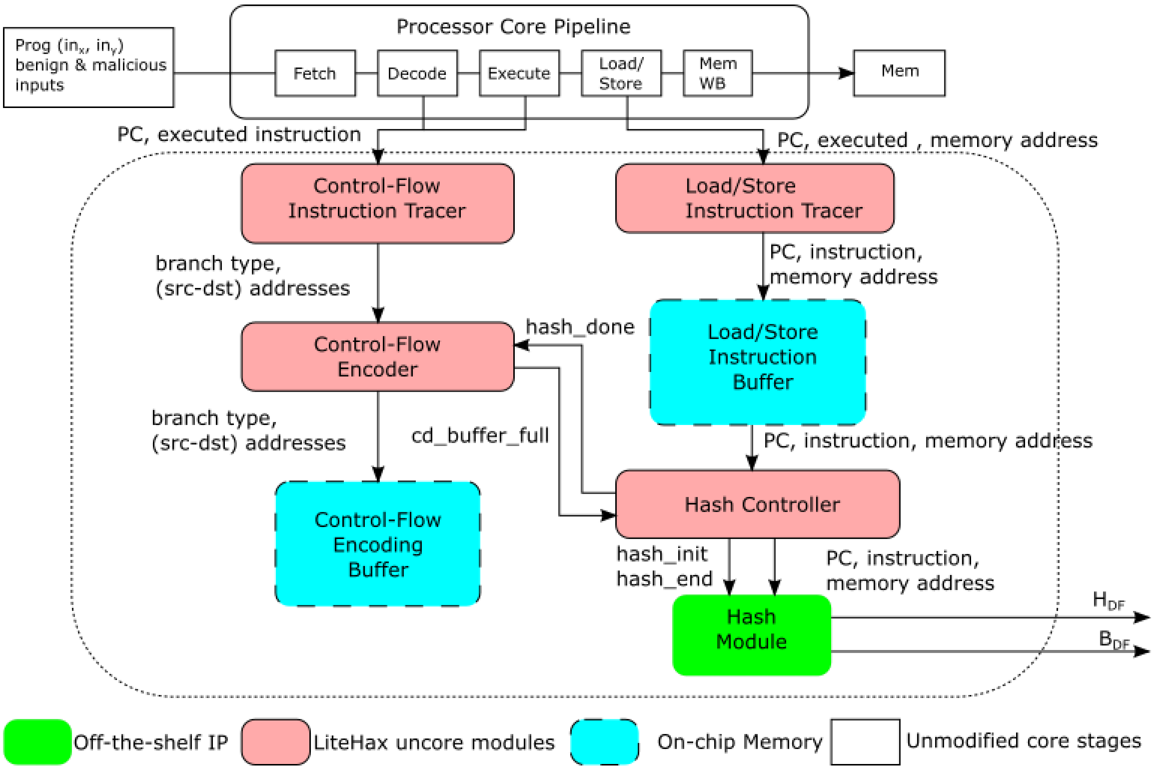


FIGURE 1.4: LiteHAX Architecture

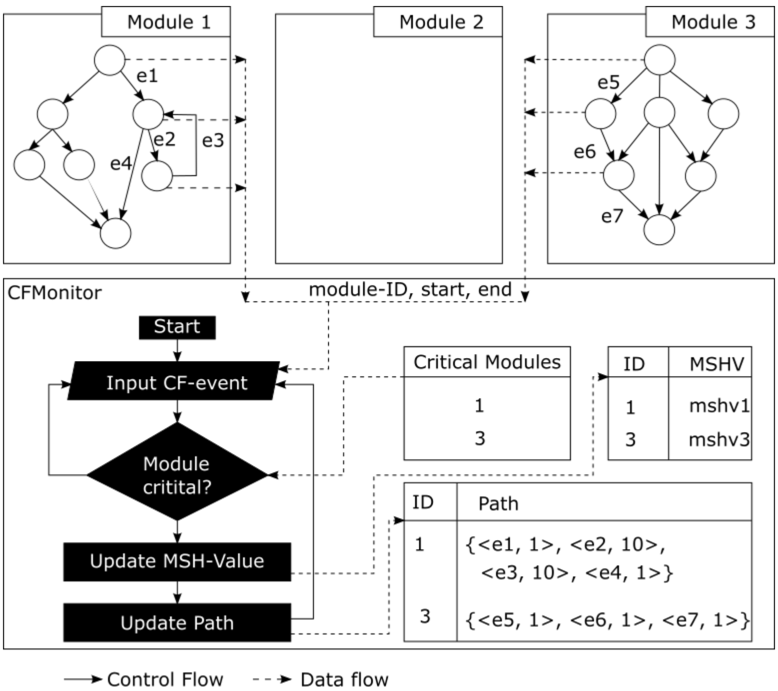


FIGURE 1.5: Diat CFMonitor Logic

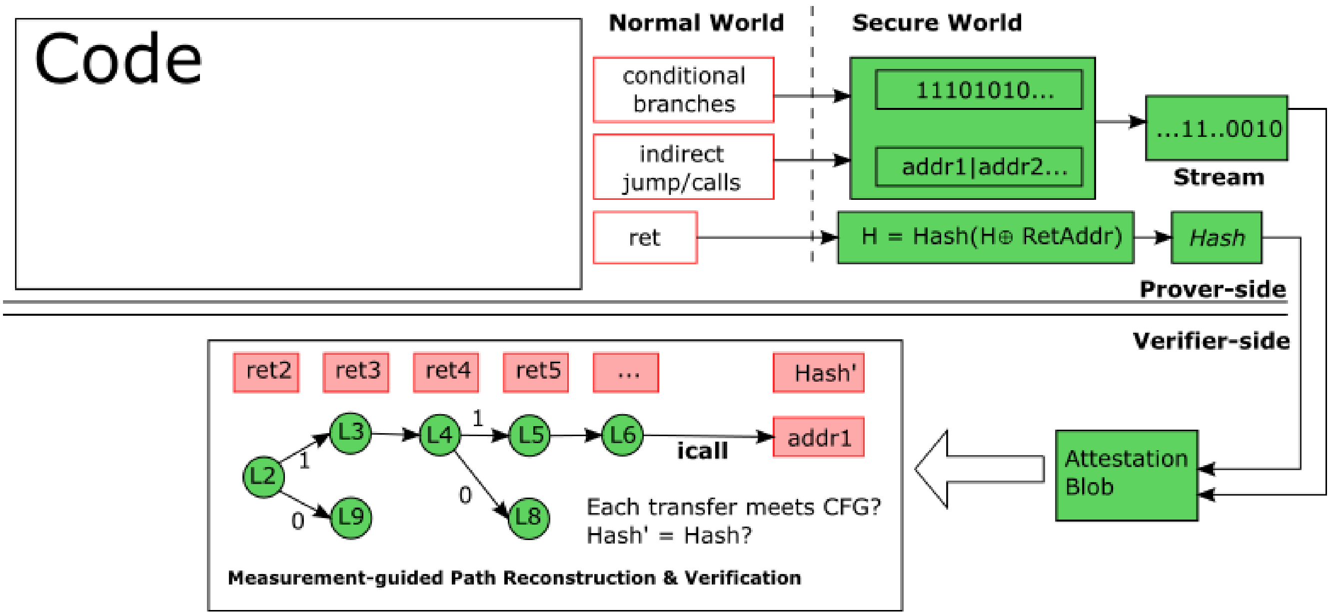


FIGURE 1.6: OAT Control-Flow Attestation

# Chapter 2

# Scope

This chapter presents the scope of the thesis. Apart of literature review, the main work on this research are the work on the models implementation for runtime remote attestation and the analysis of the model's performance. Section 2.1 presents the scope of implementation. Section 2.2 discusses the scope of the analysis.

## 2.1 Implementation

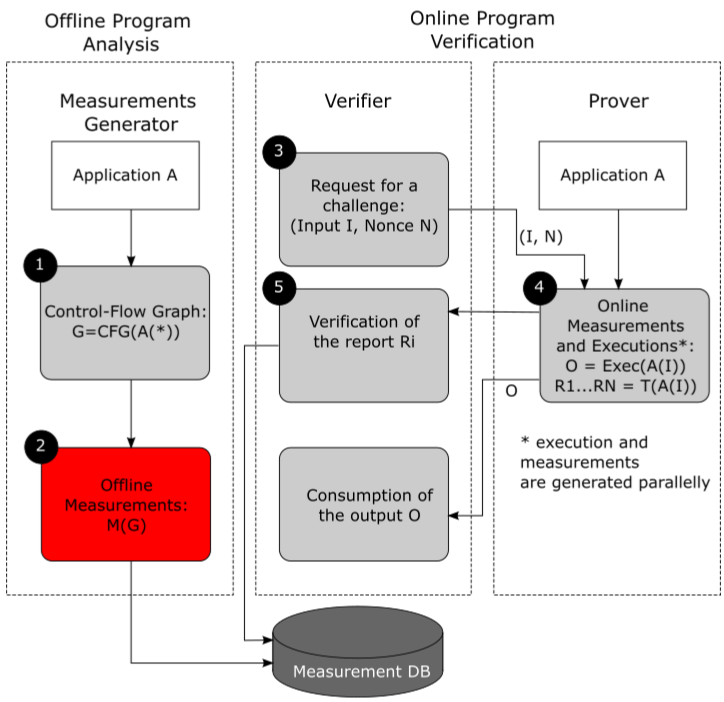


FIGURE 2.1: ScaRR System Overview

In Chapter 1, we presented different runtime remote attestation approaches. In learning the different models of runtime remote attestation, we implemented one of the model: ScaRR Control-Flow Model [29]. Specifically we write a tool to extract offline measurement using the ScaRR control-flow model. ScaRR system as shown in figure 2.1, consists of offline program analysis and online program verification. This thesis will focus on the offline program analysis part, specifically on the offline measurements generator as shown in the red box in the figure 2.1.

The offline measurement is information that will be used in runtime remote attesttion. We write the offline measurement as LLVM passes [18] using LLVM 13.0.0. We designed the algorithm based on the description in the original paper.

We tested the model implementation using different programs that is written in C. Since the LLVM pass is running against the intermediate representation (IR), we should get consistent result on any programming language that compiles to IR.

## 2.2 Analysis

We analyze the control-flow model extracted against different programs with various size and complexities. In this thesis, we are only analyzing program written in C. The analysis is comparing the program size with different measurements defined by the ScaRR control-flow model. We present the detail of the methodology in chapter 4 and show the analysis result in chapter 5.

# Chapter 3

# Background

In this chapter, we start to present brief history of memory attacks and some background information on control-flow attack in section 3.1. In section 3.2, we discuss how remote attestation helps to detect control-flow attack. We present ScaRR control flow model in section 3.3. The chapter ends in section 3.4, which share an overview of LLVM that relevant to the research.

## 3.1 Control Flow Attack

Control-flow attack happens when adversaries make a program to perform action of their choice without statically modify the program binary but alter the runtime properties of the program. The adversary intention can be to execute malicious operations or to leak secret information. Many of this runtime software security attacks are occured due to memory corruption bug in software written in low-level languages like C and C++ [28].

Once memory corruption is triggered, there are different exploit types which adversary can use to perform the attack. Some of the relevant exploits are control-flow hijack [26, 25] and data only attack [8, 7].

Control-flow hijack can be classified further into code injection attack and code reuse attack such as return oriented programming [24]. Code injection attack will inject code in the program which will execute action prepared by the attacker. Code injection attack is already mitigated by solution like non-executable stack (NX), Data Execution Prevention and $W \bigoplus R$ [30]. Code reuse attack will execute malicous action without injecting any codes, hence can't be detected by previosly mentioned defenses mechanism. As example, return-oriented program chains together short instruction sequences already present in a program's address space, each of which ends in a `return` instruction. Unfortunately, ROP can not be mitigated by $W \bigoplus R$ [24].

Memory error attacks and defenses have been always a continuous battle which unfortunately has not shown that it is over. In 2016, Abera et al proposed to use remote attestation to detect control flow attack [2]. That paper opened many researches in this area which we briefly presented in Chapter 1.

## 3.2 Remote Attestation

In this thesis we explore the use of remote attestation in detecting control-flow attack. Remote attestation is the activity of making a claim about properties of a remote target by supplying evidence to an appraiser over a network [10]. The ubiquitous deployment of IoT and different applications in the cloud require robust remote attestation method to ensure detection when the application is attacked. Remote attestation scope was only covering static attestation of the application binary. However, in the recent years there have been more sophisticated attack that can alter the behavior of application so that static attestation does not suffice.

In remote attestation, there are two roles involved, a trusted prover and a verifier. A prover is the one that must prove that the software has not been compromised. Verifier checks prover to ask the current state of runtime of the program. Alternatively, prover also can just update verifier periodically without being asked. The verifier compares the response from prover with the local database which has been generated before. If any of measurement mismatches, it means the has been violation due to an adversary's attack.

This research mainly focuses on offline measurement data generation for remote attestation which is used by verifier to validate the control flow graph. In the next section we discuss the detail of the control-flow model. We use LLVM in implementing the offline program analyzer.

## 3.3 ScaRR Control-Flow Model

ScaRR [29] are taking lesson learned from many former runtime remote attestation scheme to build model that can perform in a scalable way and can perform remote attestation on complex system. ScaRR control-flow model consists of two main components, checkpoint and list of action.

As many previous runtime attestation scheme, ScaRR models and validates the attestion based on program's control flow graph. We need to run one-time measurement computation to extract checkpoints and list of actions of the program.

### 3.3.1 Checkpoints

Checkpoint is basic block of the program that delimit execution path of the program. ScaRR defines these different checkpoint types:

- Thread Beginning: demarcating the start of program/thread
- Thread End: demarcating the end of program/thread
- Exit Point: representing exit point from application such as system call or out of translation unit function/library call
- Virtual-Checkpoint: managing cases for loop or recursion

In a program there should be at least Thread Beginning and Thread End checkpoints. Later depends on the structure of the program different checkpoint is marked in the program CFG.

### 3.3.2 List of Actions

List of actions (LoA) are edges (marked by two checkpoints) that direct one checkpoint to the next one. In program execution path, we only consider edges that identify the unique execution path.

LoA is defined through the following notation:

$$[(BBL_{s1}, BBL_{d1}), ..., (BBL_{sn}, BBL_{dn})]$$

Consider again the CFG in the Figure 4.3. The LoA between node 3 (Checkpoint Virtual) and node 10 (checkpoint ThreadEnd) is $[(BBL_3, BBL_{10})]$. However, the LoA between node 0 and node 3 is [] (empty set).

## 3.4 LLVM

LLVM is compiler framework that was developed by Chris Lattner which provides portable program representation and different tooling. LLVM supports the implementation of different frontend, backend and middle optimizer for various programming languages [**lattnerLLVMCompilationFramework2004a**].

### 3.4.1 Intermediate Representation

LLVM intermediate representation (IR) provides high-level information about programs to support sophisticated analysis and transformations. However, the representation is low-level enough to represent arbitrary programs and to allow extensive optimization. As an example, consider a simple C program in listing 3.1.

The IR of the program can be seen in listing 3.2. The text representation below, is just one of form of IR. Beside this readable instruction representation, LLVM IR also can be represented as byte code and in memory representation. In the IR, each line contains LLVM instructions. Instructions are grouped in basic blocks: container for instructions that execute sequentially. This arrangement, makes application control flow graph (CFG) to be explicit in the IR. The details of LLVM IR is available in the Language Reference [19].

LLVM optimizer — which includes Analyzer and Transformer — are working on IR. In this thesis we are using this analyzer and transformer in building the Offline Program Analyzer.

### 3.4.2 LLVM Pass

LLVM are applying transformations — which may include some analysis pipelines — and optimizations on tools called `opt`. `opt` is taking LLVM IR (either as text, bytecode or in memory) as input and then do transformations, analysis and optimizations on it (see figure 3.2). Transformation and optimization alters the LLVM structure. Analysis gets information from the structure, which usually to be used by one or more transformations. Different transformations, optimizations and analyses are performed as pipelines of LLVM passes. LLVM pass can run per function, module or loop. LLVM

```c
#include <stdio.h>

char *get_input()
{
    int rnd = rand() % 2;
    printf("get_input");
    return rnd == 1 ? "auth" : "error";
}

char *get_privileged_info()
{
    printf("get_privileged_info");
    return "you are privileged!";
}

char *get_unprivileged_info()
{
    printf("get_unprivileged_info");
    return "Invalid!";
}

void print_output(char *result)
{
    printf("%s", result);
}

void my_terminate()
{
    printf("Exiting...");
}

int main()
{
    char *access = get_input();
    char *result = "";
    if (strcmp(access, "auth") == 0)
    {
        result = get_privileged_info();
    }
    else
    {
        result = get_unprivileged_info();
    }
    print_output(result);
    my_terminate();
}
```

LISTING 3.1: Simple C Program

```
; ModuleID = 'simple-loop-no-ext.c'
source_filename = "simple-loop-no-ext.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:..." ; truncated
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i32, i32* %i, align 4
  %cmp = icmp slt i32 %0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %1 = load i32, i32* %i, align 4
  %inc = add nsw i32 %1, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %2 = load i32, i32* %retval, align 4
  ret i32 %2
}

attributes #0 = { noinline nounwind uwtable ... } ; truncated ...

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

LISTING 3.2: LLVM IR The Sample C Program

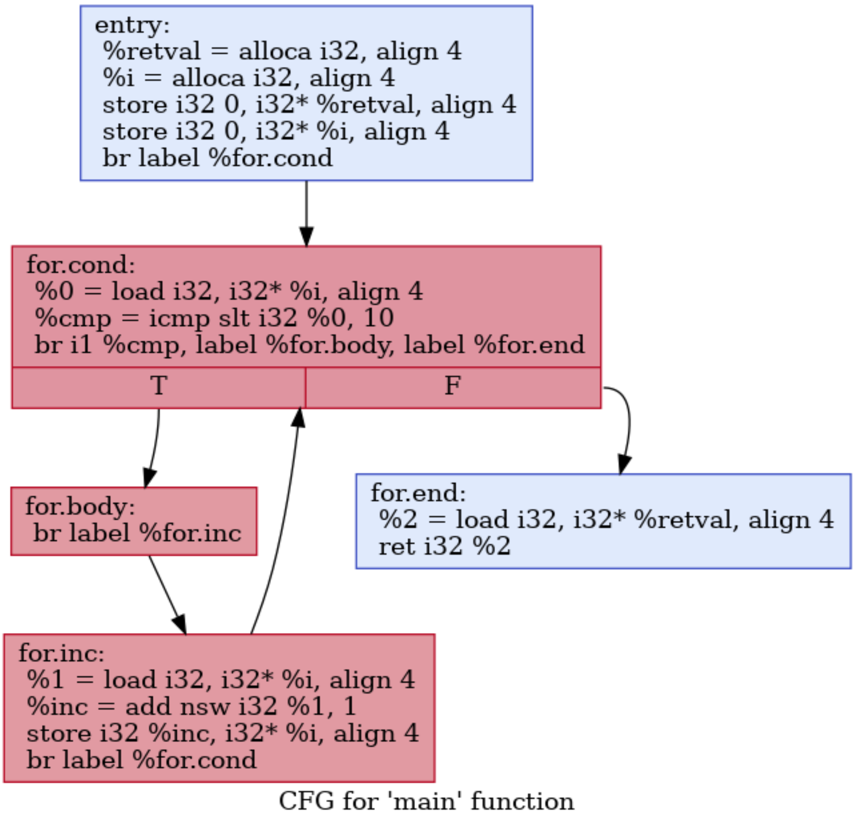


FIGURE 3.1: CFG for Simple C Program

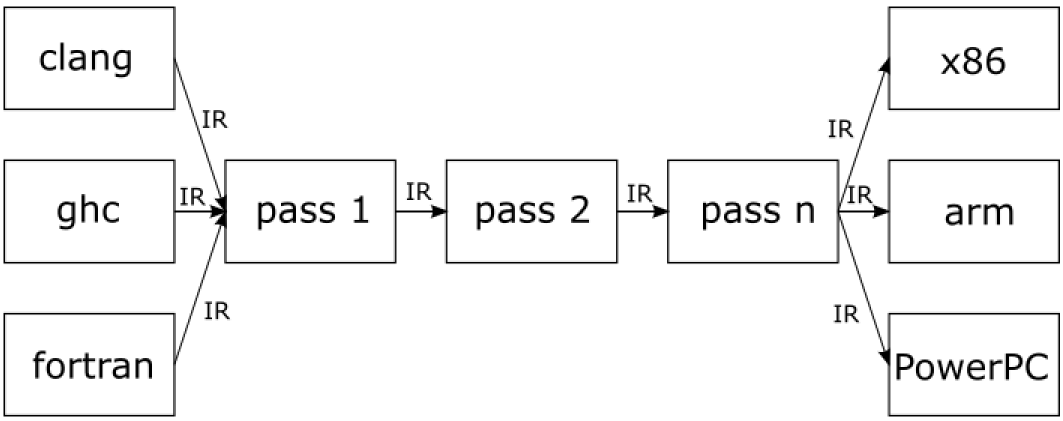


FIGURE 3.2: LLVM Pass

function pass is executed once for every function in the program. LLVM module pass is executed once for every module. LLVM loop pass runs a time for each loop.

In LLVM there are two ways of implementing Pass. First is using legacy approach and the latest one is using new pass manager approach. The approach different in structuring the code the implement the pass and also the way we use the pass. In the legacy approach, we need to inherit from either `ModulePass`, `FunctionPass` or `LoopPass` and override `runOnXXX` method (xxx is either Function, Module or Loop). In the newer approach we have to inherit CRTP mix-in `PassInfoMixin<PassT>` and override the run method.

The way we use the pass, in legacy approach we need to provide the pass name as literal argument to `opt`. See the example in listing 3.3. In the new pass manager, we are putting the pass name after '–passes' argument in comma separated list (listing 3.4). The pass is executed in order.

```
opt --dot-cfg file.ll
```

LISTING 3.3: Running Legacy LLVM Pass

```
opt -passes=scarr-cp-marker,scarr-loa-collector file.ll
```

LISTING 3.4: Running LLVM New Pass

### 3.4.3 LLVM API

In writing LLVM pass, we use LLVM API. In this section we present relevant component that is required in implementing LLVM Pass for the Offline Program Analyzer. LLVM API is leveraging many C++ features and libraries such as template and STL. The API also provides many ready to use data structure which is not available in the STL. A more broad discussion on the important element of the API is available in the Programmers Manual [21]. Complete API documentation can be referred at the doxygen page [20].

#### Module

Module is the top level container for all other IR objects. Module contains list of global variables, functions, symbol tables and other various data about target characteristics. Module can be a single translation unit of a program (source file) or can be multiple translation unit combined by linker.

In LLVM pass, we can get access to module by implementing a Module pass or by parsing IR using `parseIR` or `parseIRFile` from `IRReader.h`. Once we get a handler to a module, getting a functions within module is as simple as passing module to a loop, since module provides iterator that return list of function in the module (see listing 3.5).

```cpp
#include <llvm/IR/LLVMContext.h>
#include <llvm/IR/Module.h>
#include <llvm/IRReader/IRReader.h>
#include <llvm/Support/SourceMgr.h>

int main()
{
    LLVMContext ctx;
    SMDiagnostic Err;
    auto module = parseIRFile("ir-file.ll", Err, ctx);
    for (auto &function : *module)
    {
        // do thing with function
    }
}
```

LISTING 3.5: LLVM Module API

### Function

Function in LLVM represents function in the source program. A function contains list of zero or more BasicBlocks. There is one entry BasicBlock and can be multiple exit BasicBlocks. We can get handler to a function either by getting the iterator from a module instance or by implement a Function Pass. By using optimization, syntax hint or using an inliner pass, a function can be inlined. In this thesis, we are using *inliner-wrapper* pass to inline most function before feeding the IR into the ScaRR passes.

### Basic Block

Basic Block represents single entry and single exit section of the code. The single exit can be one of terminator instruction — branches, return, unwind and invoke. We can get handle to a basic block from function. Refer to listing 3.6 to see how to get the basic block.

```cpp
        for (auto &function: *module) {
            for (auto &basicBlok: function) {
                // do thing with Basic Block
            }
        }
```

LISTING 3.6: LLVM Basic Block API

### Graph Traversal

Since LLVM CFG is already structured as a graph, the basic block can be traversed using different ready to use graph traversal algorithm. LLVM offers some common graph traversal algorithms such as breadth first search and depth first search. The algorithms can be used immediately on basic blocks and functions. If there is a need to traverse a custom structure, the algorithms just require the new structure to implement *GraphWriter* interface.

## 3.4.4 Tools

We are implementing the algorithms using different tools. We are highlighting some of those in this section so that everyone interested can replicate the step.

### clang

`clang` is one of the frontend provided by LLVM. It can compiles C, C++ and Objective C. `clang` command line arguments are compatible with widely use gcc compiler. The main use of `clang` in this research is to compile source files into LLVM IR text files. Listing 3.7 shows how to compile a C program into LLVM IR.

```
clang -S -emit-llvm source.c
```

LISTING 3.7: Compiling C to LLVM IR

We can pass optimization level from 0 (no-optimization) to 3 (most optimal, can make code run faster but larger in size) when compiling the source code.

By default, clang strips out value names and do some optimization when generating LLVM IR. We can use this flag to disable optimization and get readable value names that can help when troubleshooting and exploring the generated IR. Listing 3.8 shows how to compile to IR without any optimization and to preserve the function and variable names.

```
clang -S -emit-llvm -Xclang -disable-O0-optnone \
-fno-discard-value-names source.c
```

LISTING 3.8: Compiling C to LLVM IR without Optimization

### opt

`opt` is LLVM optimizer and analyzer that can be invoked from command line. We are using `opt` to execute the offline program analyzer which marks basic block checkpoints and calculate list of action which can be used as information to detect control flow violation during remote attestation.

### cmake

`cmake` is a build file generator which is have an important role in large project like LLVM. Although the deep understanding of `cmake` is not required in implementing LLVM pass, but we need to know at least how to build the pass after the implementation so that we can run it.

LLVM can be downloaded using git. See Listing 3.9. This thesis is implemented on LLVM 13.0.0.

```
git clone https://github.com/llvm/llvm-project
```

LISTING 3.9: Cloning LLVM Source Code

Once it is downloaded we can go to the LLVM directory and generate the build files. `cmake` supports several build tools such as make and ninja. Refer to listing 3.10.

```
cd llvm-project/llvm
mkdir build
cd build
cmake -G Ninja ../ # generate build file for Ninja
ninja opt # build only opt
```

LISTING 3.10: Building LLVM

With all background discussion in this chapter we should be ready to discuss the methodology in Chapter 4.

# Chapter 4

# Methodology

In this chapter, we present the methodology of the research. First we present the threat model in section 4.1. After that, we show the overview of ScaRR algorithm to extract checkpoint and list of action in section 4.2. Section 4.3 discusses the LLVM implementation to get checkpoints. Section 4.4 discusses on methodology of getting list of actions. Checkpoints and list of actions are collected to build offline measurement database that is used for the remote attestation. The last section shows how to run the LLVM passes to get the result which is presented in Chapter 5.

## 4.1 Threat Model

The threat model in this research is taken from ScaRR [29]. There are two parties: attacker and prover.

**Attacker capabilities:** The attacker aims to control remote service using various method such as memory attack or any attack in user-space. The attacker has bypassed memory attack protection such as Control Flow Integrity (CFI) or $W \bigoplus R$ or ASLR using techniques like Return-Oriented Programming (ROP)[24] or Jump-Oriented Programming (JOP) [5]. We do not consider physical attack and also non-control data attack which does not alter program's CFG.

**Defender capabilities:** The prover uses kernel as trusted anchor and has common memory corruption attack mitigations such as $W \bigoplus R$ and ASLR.

## 4.2 Overview of The Offline Measurement


FIGURE 4.1: Generating Offline Measurement

The goal of the offline measurement is to get the information to be used in the remote attestation. In this research we implement ScaRR Control-flow model [29] which we also elaborate in the Section 3.3. Figure 4.1 shows the step on generating the offline measurement.

We start with flattening the CFG into basic block. The CFG is structured as graph that can contains branches and cycles. The flattening process will traverse each and every graph once into a list of basic block.

Consider the program in listing 4.1. The CFG is shown in the figure 4.2. The first step of the offline measurement is to flatten the CFG to get all the basic block list. We can flatten the CFG into basic block with one out of some graph traversal algorithm in LLVM. The result of flattened graph is in the right side of figure 4.2

```
int main() {
    for (int i = 0; i < 10; i++) {
        // do nothing
    }
}
```

LISTING 4.1: Simple Loop

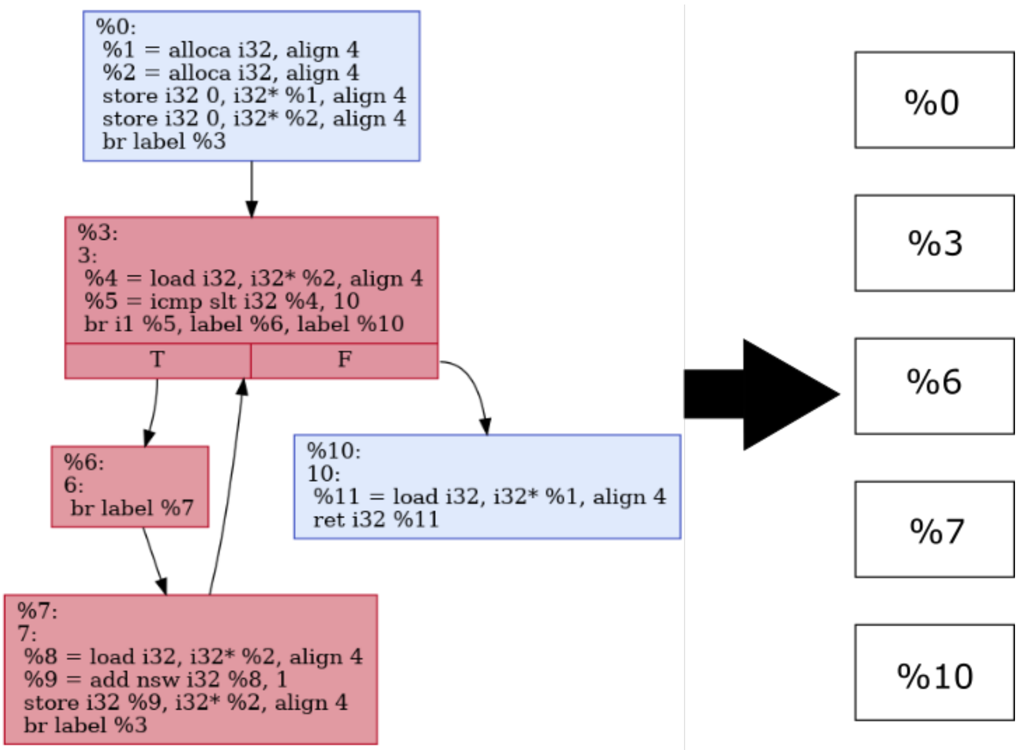


FIGURE 4.2: Simple Loop CFG

After flattening the CFG we will have to pass the CFG for two times. First, to mark the checkpoints (section 4.3). Second, to find all list of actions between the checkpoint (section 4.4).

Checkpoints and list of actions are used to for offline measurements which is represented as triplet of first and second checkpoint; and hash of the list of actions.

$$(cp_A, cp_B, H(LoA)) \Rightarrow [(BBL_{s1}, BBL_{d1}), ..., (BBL_{sn}, BBL_{dn})]$$

The offline measurement is consulted by verifier during remote attestation.

## 4.3 ScaRR Checkpoint Marker

After we flatten CFG into basic block, we will check each of basic block to find different kind of checkpoints. In Chapter 3 we list these four types of checkpoints: **Thread Begin**, **Thread End**, **Exit Point** and **Virtual Checkpoint**. Now we will present the heuristic on how we will mark as a checkpoint when appropriate. The logic of checkpoint marker is to traverse the whole control flow graph at least once. For each basic block, we have to check whether the basic block can be considered as any of checkpoint type mentioned above. To allow marking additional information about ScaRR checkpoint, we are modifying the BasicBlock class to add checkpoint instance variable as shown in listing 4.2.

```
class BasicBlock ... {
private:
// add checkpoint field
    Checkpoint cp;

public:
    // setter and accessor
void setCheckpoint(Checkpoint);
Checkpoint getCheckpoint() const;
...
}
```

LISTING 4.2: Add Checkpoint Instance Variable to BasicBlock class.

Thread begin identifies the beginning of a thread or start of program. In this thesis we mark this checkpoint to first basic block in main function. If a program is is a multithreaded program, we will mark the thread begin for each basic block that starts the thread.

In the other side, thread end marks the end of a thread or end of program. In a multiple-threaded program, we will mark the thread end for each of basic block that terminates a thread. In this thesis, we mark thread end checkpoint for last basic block that has no more successors.

Exit point marks that a basic block is calling function outside of translation unit. The heuristic of marking this type of basic block is we iterate all instructions in a basic block. For each instruction, if the instruction is a `call` instruction, we check whether the called function has any basic block. If the function has no basic block, it means it is an external function, hence we will mark this as an exit point and stop. If none of instruction is a `call` instruction or all `call` instruction in this basic block call function with non empty basic block, it means this basic block calls internal function, therefore this basic block is not an exit point. Please refer to Listing 4.3.

```
for (auto &basicBlock: Function) {
    for (auto &instruction : basicBlock) {
        if (isa<CallInst>(i)) {
        auto *call = &cast<CallBase>(i);
        if (call != nullptr && call->getCalledFunction()->empty()) {
            // this basicBlock is ExitPoint
            basicBlock.setCheckpoint(Checkpoint::ExitPoint);
        }
    }
}
```

LISTING 4.3: Finding ExitPoint Checkpoint

Virtual checkpoint is a checkpoint that marks special cases such as loop or recursion. We will discuss only for loop case in this thesis. Virtual checkpoint in a loop is basically a loop header. The heuristic to find a loop header is to use `DominatorTree` to find a loop. After we find a loop, then we just need to get the header. Although there is no direct API to check whether a basic block is a loop header, LLVM provide it in LoopInfoBase API. See Listing 4.4

```
void findVirtualCheckpoint(DominatorTree &DT, Function &F) {
    DT.recalculate(F);
    // generate the LoopInfoBase for the current function
    LoopInfoBase<BasicBlock, Loop>* KLoop = new LoopInfoBase<BasicBlock, Loop>();
    KLoop->releaseMemory();
    KLoop->analyze(DT);
    for (auto &bb : F) {
        // Since the BasicBlock would have been inlined, just traverse from main functio
        if (F.getName() == "main") {
        auto loop = KLoop->getLoopFor(&bb);
        if (loop != nullptr) {
            // found VirtualCheckpoint
                loop->getHeader()->setCheckpoint(Checkpoint::Virtual);
        }
        }
    }
}
```

LISTING 4.4: Getting Virtual Checkpoint

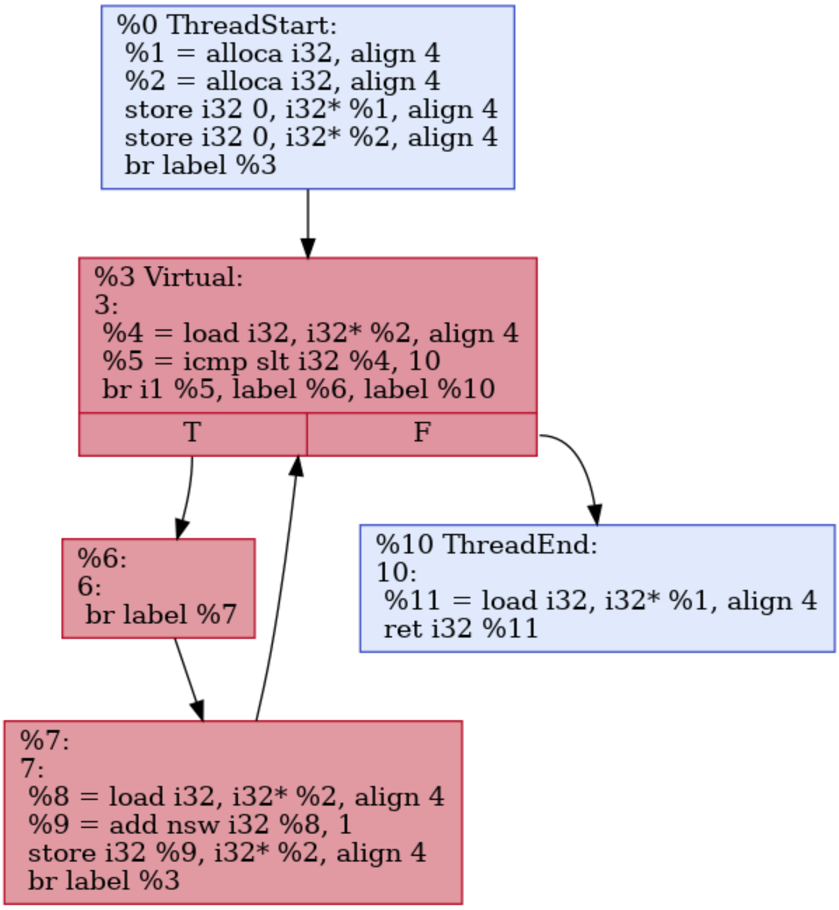


FIGURE 4.3: Loop CFG

## 4.4 ScaRR LoA Collector

After we mark checkpoints in the CFG, now we can find list of actions. The step of finding LoA is traversing path between two checkpoints and add significant basic block that traverse the path between the two checkpoints. The detail of this step is explained in section 4.4.

Jon TODO: summarize the algorithm in a pseudocude fashion? you should link the lines in the pseudo code and describe it step by step. ◁

```
// TBD
```

LISTING 4.5: TBD Pseudocode for LoA

The algorithm of getting LoA between two checkpoints is little bit more complex. First, we iterate all the basic block and if the basic block is a checkpoint we mark this is $cpA$. Next, we recursively traverse the successor of $cpA$ until we find another checkpoint $cpB$. It is possible for $cpA = cpB$. If there is no branch between the two checkpoint, the LoA is an empty set. If there is a branch, the first LoA is always be $cpA$ and the second LoA is always be the first basic block after the branch — which can be $cpB$ or just non checkpoint basic block. Interested readers can refer to the implementation of this pass to see the detail.

## 4.5 Running The Pass

To mark the list of Checkpoints, we can invoke LLVM `opt` as shown in listing 4.6.

```
opt -passes=scarr-cp-marker <file>.ll
```

LISTING 4.6: Mark Checkpoint in BasicBlock

We can see the basic blocks output that has been marked with checkpoint using LLVM dot-cfg pass.

```
opt -passes=scarr-cp-marker,dot-cfg <file>.ll
```

LISTING 4.7: Print Checkpoints in CFG dot file

The commands in listing 4.7 generates different dot files per function. We can use xdot command line from graphiz to see the graph.

To mark the list of actions between checkpoints, we can invoke LLVM `opt` as shown in Listing 4.8

Note that we have to run scarr-cp-marker before scarr-loa-collector.

The result and its interpretation are discussed in the next chapter.

```
opt -passes=scarr-cp-marker,scarr-loa-collector <file>.ll
```

LISTING 4.8: Get List of Actions

# Chapter 5

# Results

In this research we used the offline measurement generator in getting the measurement across different real world programs. We calculate the ScaRR control flow information for each of the programs and we analyse and present the result in this chapter. The analysis and source code of the program is available in this Github repository: https://github.com/lamida/scarr-sample-program/.

In this research we choose 4 large open source projects for analysis. We download the source code and checkout the latest version of the code. The first software is Redis 6.2.4 [23], a what so called data structure server that is widely used in real world. The Redis source build consist of the server binary and the client cli. We analyse both of the program.

The second program we analyse is bzip2 1.08 [6]. Bzip2 is free and open source file compression program. The third program is openssl $1.1.1j$ [22]. OpenSSL is full-featured toolkit for TLS protocol.

The last suite program we analyse is coreutils 8.32 [11]. Coreutils is suite of Unix utilities for file, shell and text manipulation.

For each of program, we are collecting the following measurements:

- source code lines
- IR lines
- number of basic blocks (nBB)
- number of ScaRR measurements (nM)
- number of checkpoints (nCP)
- number of LoA (nLoA)

## 5.1 ScaRR Control Flow Result

Jon Elaborate the results, add charts for better visualization than just table. ◁ Jon TODO: remove this long table and use simpler visualization ◁

| program | code lines | IR lines | nBB | nM | nCP | nLoA |
|---|---|---|---|---|---|---|
| basename | 190 | 827 | 19 | 17 | 13 | 26 |
| cat | 767 | 2215 | 119 | 151 | 94 | 216 |
| chgrp | 319 | 1158 | 43 | 44 | 32 | 60 |

| program | code lines | IR lines | nBB | nM | nCP | nLoA |
|---|---|---|---|---|---|---|
| cksum | 310 | 1012 | 9 | 11 | 8 | 20 |
| cp | 1226 | 3619 | 79 | 37 | 27 | 56 |
| csplit | 1526 | 5550 | | | | |
| cut | 609 | 2196 | 40 | 36 | 25 | 56 |
| date | 604 | 2223 | 69 | 80 | 57 | 110 |
| dd | 2581 | 9420 | 366 | 451 | 235 | 748 |
| df | 1847 | 8146 | 377 | 410 | 278 | 698 |
| dirname | 136 | 635 | 13 | 13 | 10 | 20 |
| du | 1140 | 3973 | 206 | 273 | 138 | 424 |
| env | 952 | 3875 | 197 | 241 | 141 | 374 |
| expand | 238 | 1057 | 46 | 55 | 37 | 78 |
| expr | 1117 | 5549 | | | | |
| false | 2 | 442 | 6 | 7 | 6 | 12 |
| fmt | 1029 | 4083 | 44 | 45 | 26 | 64 |
| getlimits | 172 | 2681 | 109 | 189 | 109 | 324 |
| head | 1095 | 3596 | 196 | 253 | 149 | 386 |
| id | 464 | 1922 | 67 | 47 | 31 | 84 |
| install | 1059 | 3545 | 120 | 129 | 72 | 162 |
| kill | 314 | 1376 | 64 | 90 | 49 | 142 |
| link | 93 | 581 | 10 | 8 | 8 | 12 |
| ln | 681 | 2350 | 64 | 60 | 37 | 100 |
| ls | 5520 | 24925 | 394 | 450 | 249 | 654 |
| make-prime-list | 230 | 813 | 32 | 43 | 27 | 78 |
| mkdir | 296 | 1230 | 28 | 30 | 21 | 42 |
| mktemp | 350 | 1347 | 65 | 80 | 53 | 120 |
| mv | 512 | 1903 | 66 | 61 | 40 | 92 |
| nice | 221 | 905 | 27 | 31 | 24 | 54 |
| nl | 596 | 1994 | 35 | 39 | 24 | 58 |
| numfmt | 1651 | 6036 | 113 | 118 | 69 | 154 |
| od | 1987 | 7876 | 230 | 291 | 173 | 454 |
| paste | 530 | 2234 | 31 | 41 | 27 | 54 |
| pathchk | 422 | 1306 | 56 | 77 | 44 | 120 |
| pinky | 602 | 3161 | 72 | 84 | 48 | 128 |
| pr | 2848 | 10596 | 105 | 89 | 50 | 142 |
| printenv | 154 | 742 | 25 | 34 | 20 | 60 |
| printf | 715 | 2811 | 118 | 147 | 92 | 200 |
| ptx | 2153 | 7888 | 487 | 651 | 294 | 1060 |
| pwd | 394 | 1777 | 65 | 74 | 50 | 116 |
| readlink | 178 | 813 | 34 | 36 | 24 | 50 |
| realpath | 278 | 1382 | 74 | 86 | 49 | 134 |
| rm | 373 | 1213 | 40 | 41 | 26 | 64 |
| rmdir | 253 | 1048 | 31 | 38 | 23 | 62 |
| seq | 736 | 3057 | 105 | 132 | 84 | 226 |
| shred | 1279 | 3789 | 67 | 83 | 52 | 130 |
| shuf | 615 | 2524 | 128 | 177 | 106 | 310 |
| sleep | 146 | 688 | 21 | 28 | 18 | 46 |

| program | code lines | IR lines | nBB | nM | nCP | nLoA |
|---|---|---|---|---|---|---|
| split | 1668 | 6262 | 248 | 302 | 189 | 510 |
| stat | 1907 | 18653 | 55 | 65 | 44 | 86 |
| stdbuf | 394 | 1644 | 87 | 109 | 76 | 154 |
| stty | 2322 | 5885 | 163 | 167 | 110 | 250 |
| sum | 273 | 1312 | 15 | 19 | 14 | 32 |
| sync | 239 | 838 | 29 | 39 | 27 | 58 |
| tac | 713 | 2324 | 64 | 75 | 48 | 116 |
| tail | 2537 | 8657 | 458 | 575 | 329 | 894 |
| tee | 278 | 1193 | 48 | 61 | 37 | 96 |
| test | 867 | 3968 | | | | |
| tr | 1914 | 6139 | 152 | 167 | 97 | 282 |
| true | 80 | 441 | 6 | 7 | 6 | 12 |
| truncate | 388 | 1564 | 78 | 94 | 58 | 150 |
| tty | 133 | 624 | 13 | 14 | 12 | 20 |
| uname | 376 | 1236 | 76 | 101 | 61 | 140 |
| unexpand | 326 | 1255 | 60 | 67 | 46 | 98 |
| uniq | 662 | 2379 | 110 | 125 | 77 | 192 |
| unlink | 88 | 543 | 7 | 5 | 6 | 10 |
| uptime | 257 | 1249 | 5 | 2 | 3 | 4 |
| users | 150 | 899 | 5 | 2 | 3 | 4 |
| wc | 895 | 3689 | 89 | 100 | 68 | 158 |
| yes | 130 | 765 | 19 | 26 | 15 | 44 |

Jon Find a way to add caption to this long table ◁

| program | code lines | IR lines | nBB | nM | nCP | nLoA |
|---|---|---|---|---|---|---|
| redis-benchmark | 1982 | 29367 | 254 | 407 | 211 | 564 |
| redis-cli | 8400 | 45836 | 513 | 630 | 388 | 1002 |
| server | 6397 | 25790 | 88 | 91 | 52 | 138 |

TABLE 5.2: Redis ScaRR measurements

| program | code lines | IR lines | nBB | nM | nCP | nLoA |
|---|---|---|---|---|---|---|
| bzip2 | 2036 | 8542 | 143 | 132 | 76 | 244 |
| openssl | 832 | 1793 | 28 | 30 | 19 | 42 |

TABLE 5.3: Some additional programs ScaRR measurements

## 5.2 Complexity Analysis

TBD

## 5.3 Case Study

TBD

# Chapter 6

# Discussion and Future Works

TBD

# Chapter 7

# Conclusion

In this thesis we implemented ScaRR control flow model extractor that can be used to build offline measurement database. We We presented the design and the implementation of the tool as two different LLVM passes.

TBD

# Bibliography


[1] Martín Abadi et al. "Control-Flow Integrity Principles, Implementations, and Applications". In: (2005).

[2] Tigist Abera et al. *C-FLAT: Control-Flow Attestation for Embedded Systems Software*. Tech. rep. 2016.

[3] Tigist Abera et al. "DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems". In: (2019). DOI: 10.14722/ndss.2019.23420.

[4] Arash Baratloo. *Transparent Run-Time Defense Against Stack Smashing Attacks*. Tech. rep. 2000.

[5] Tyler Bletsch et al. *Jump-Oriented Programming: A New Class of Code-Reuse Attack*. 2011. ISBN: 978-1-4503-0564-8.

[6] *Bzip2 / Bzip2*. en. https://gitlab.com/bzip2/bzip2.

[7] Nicolas Carlini et al. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity". en. In: (2015), p. 17.

[8] Shuo Chen et al. *Non-Control-Data Attacks Are Realistic Threats*. Tech. rep. 2005.

[9] Dwaine Clarke et al. "Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking". en. In: *Advances in Cryptology - ASIACRYPT 2003*. Ed. by Gerhard Goos et al. Vol. 2894. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 188–207. ISBN: 978-3-540-20592-0 978-3-540-40061-5. DOI: 10.1007/978-3-540-40061-5_12.

[10] George Coker et al. "Principles of Remote Attestation". en. In: *International Journal of Information Security* 10.2 (June 2011), pp. 63–81. ISSN: 1615-5262, 1615-5270. DOI: 10.1007/s10207-011-0124-7.

[11] *Coreutils/Coreutils*. coreutils. July 2021.

[12] Ghada Dessouky et al. "LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution". In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD* (2018). ISSN: 10923152. DOI: 10.1145/3240765.3240821.

[13] Ghada Dessouky et al. *LO-FAT: Low-Overhead Control Flow ATtestation in Hardware*. Tech. rep. 2017.

[14] Vivek Haldar, Deepak Chandra, and Michael Franz. *Semantic Remote Attestation-A Virtual Machine Directed Approach to Trusted Computing*. Tech. rep. 2004.

[15] Chongkyung Kil et al. *Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software*. Tech. rep. 2006.

[16] Florian Kohnhäuser et al. "SCAPI: A Scalable Attestation Protocol to Detect Software and Physical Attacks". en. In: *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. Boston Massachusetts: ACM, July 2017, pp. 75–86. ISBN: 978-1-4503-5084-6. DOI: 10.1145/3098243.3098255.

[17] Nikos Koutroumpouchos et al. "Secure Edge Computing with Lightweight Control-Flow Property-Based Attestation". In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. June 2019, pp. 84–92. DOI: 10.1109/NETSOFT.2019.8806658.

[18] Chris Arthur Lattner. "LLVM: AN INFRASTRUCTURE FOR MULTI-STAGE OPTIMIZATION". PhD thesis. 2002.

[19] *LLVM Language Reference Manual — LLVM 13 Documentation*. https://llvm.org/docs/LangRef.html.

[20] *LLVM: LLVM*. https://llvm.org/doxygen/.

[21] *LLVM Programmer's Manual — LLVM 13 Documentation*. https://llvm.org/docs/ProgrammersManual.

[22] *Openssl/Openssl*. OpenSSL. July 2021.

[23] *Redis/Redis*. Redis. July 2021.

[24] Ryan Roemer et al. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur* 15.2 (2012). DOI: 10.1145/2133375.2133377.

[25] Felix Schuster et al. "Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications". en. In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2015, pp. 745–762. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.51.

[26] Hovav Shacham. *The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)*. Tech. rep. ACM Press, 2007.

[27] Zhichuang Sun et al. "OAT: Attesting Operation Integrity of Embedded Devices". In: *2020 IEEE Symposium on Security and Privacy (SP)*. May 2020, pp. 1433–1449. DOI: 10.1109/SP40000.2020.00042.

[28] László Szekeres et al. *SoK: Eternal War in Memory*. Tech. rep. 2013.

[29] Flavio Toffalini et al. *ScaRR: Scalable Runtime Remote Attestation for Complex Systems*. Tech. rep. 2019.

[30] Victor Van Der Veen et al. *Memory Errors: The Past, the Present, and the Future*. Tech. rep. 2012.

[31] Shaza Zeitouni et al. "ATRIUM: Runtime Attestation Resilient under Memory Attacks". In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD* 2017-Novem (2017), pp. 384–391. ISSN: 10923152. DOI: 10.1109/ICCAD.2017.8203803.