SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

Study of Models for Runtime Remote Attestations

Submitted by

Jon Kartago LAMIDA

Thesis Advisor

FLAVIO TOFFALINI and Prof. JIANYING ZHOU

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)

August 17, 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, Chapter 103: 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 attestations allow one to detect control-flow attacks that can alter the intended behavior of a program. The Scalable Runtime Remote Attestation for Complex Systems (ScaRR) introduced runtime remote attestation on complex systems [26]. Before that, runtime remote attestation was only feasible for embedded systems.

This thesis presents the implementation of ScaRR offline measurement using LLVM and analyzes the performance of the computation and verify the scalability of the algorithm. We discuss related runtime remote attestation schemes with their features and limitations in the beginning part of the thesis. We also present relevant background on the control-flow attack, remote attestation, and LLVM. We analyze ScaRR control-flow model before jump into the implementation and testing the model.

At the end of this research, we show that the ScaRR control-flow model has linear time and space complexity. This complexity is significantly better than most runtime remote attestations that require exponential complexity for their model representation and verification. ScaRR is scalable and usable for large program remote attestation.

Acknowledgements

I would like to thank my advisors Flavio Toffalini and Prof. Jianying Zhou for their guidance during this research and the thesis writing. We started the work almost nine months ago. From then, I have learned so many new things that excite me a lot. First, I learned the scientific research process. Second, I have learned many new areas of interest and subjects related to the thesis topic, from LLVM, modern C++, compiler, computer architecture, and software security. All of the learning will become important for my future professional careers and probably a trigger for another journey in graduate school.

This thesis also would not have been possible without the support of my family. I have taken many hours of family time from my wife and my daughter for me to finish this project. My wife inspires me to go to graduate school. Now I want to inspire my children on long-life learning too. Finally, I also want dedicate this project to my parents and my siblings. I will not be here now if not because of them.

Contents

List of Figures

List of Tables

List of Listings

Dedicated to my parents, my lovely wife, my daughter, and my soon to born son.

Chapter 1

Introduction

Memory corruption errors have been a major security problem for almost 40 years [25, 27]. The errors often happen due to the use of unsafe languages, mainly C and C++. This safety issue becomes a significant attack surface that affects the security of many critical applications. The adversaries can hijack the program and taking control to achieve their goals. Such an attack impacts a wide range of applications, from distributed systems running in the cloud to embedded programs installed on IoT devices.

Since the identification of these vulnerabilities, continuous arm races between adversaries and defenders have been happening. As attackers find a vulnerability, then different defense mechanisms are invented. After that, attackers and researchers find another vulnerability that can circumvent or disable the defense. Or sometimes, the solutions are not just practical to be deployed. So far, we deploy some solutions such as non-executable stack (NX) or Data Execution Prevention (DEP) $W \bigoplus R$ [27] Control Flow Integrity (CFG) [1], ASLR [15], Stack Canaries [4] and more. Unfortunately 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 attacks on IoT or other embedded devices. Since then, many runtimes remote attestation schema are introduced [13, 28, 16, 12, 3, 17, 24]. Like C-Flat, most of those remote attestation schemes are targeting 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 [26].

This thesis studies different models for runtime remote attestation. We focus on the ScaRR implementation due to the unique strength in making the attestation work for a 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

We discuss different runtime remote attestation models in this section. Specifically, we discuss how different attestation schemes encode the offline program representations.

C-Flat [2] is the first remote attestation scheme to detect runtime control flow attack for embedded systems. C-Flat generates offline measurement by traversing all program's possible paths, from the start node to the termination node. In each node, C-Flat hashes the node ID and the hash of the previous node. In the first node, since there is no previous hash, we set the hash to 0. All of those steps create hash chains which we store as an offline measurement in a database.

C-Flat can attest exact flow path of a program from the hash chains information. C-Flat does not 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— on its inefficiencies for having to explore all possible paths from program control flow graph [2].

Lo-Fat [13] improves C-Flat by using hardware support for control flow attestation. The hardware intercepts the instructions and process them 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 induces high verification cost.

Atrium [28] is a remote attestation scheme that can provide resiliency against physical memory attacks where adversaries can exploit the property of Time of Check Time of Use (TOCTOU) during attestation. In the paper, the author describes memory bank attacks. In that attack, the adversary can control instruction fetches to prevent detection. During attestation time, the program fetches the instruction from the benign memory area; when attestation does not run, the attack directs the fetch to the malicious area.

Atrium calculates the offline program measurement differently compared with C-Flat and Lo-Fat. In Atrium, the verifier performs a one-time process to generate the CFG of the program. Then, Atrium computes hash measurement on the instructions and addresses of basic blocks. As a comparison, C-Flat only hashes the node ID. While this different approach can mitigate the TOCTOU attack, the offline measurement generation still grows exponentially as the complexity of the program increases.

LiteHax [12] is hardware assisted remote attestation scheme for RISC-V embedded devices 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.

The offline measurement phase of LiteHax only generates program CFG without calculating any hash over the whole control flows and data flow events. However, during verification, provers still send the computed hashes as report to the verifier. Verifier processes the report in two steps. First, it executes symbolic execution and then runs an incremental forward data-flow analysis. The two steps do not require any lookup to offline measurement database.

Diat [3] is a remote attestation scheme that can attest data integrity and control-flow of autonomous collaborative network systems. The program attested must be decomposed into small interacting modules. The scheme decomposes a program into different modules to improve the efficiency of the attestation. Diat sets data-flow monitoring between critical modules. Diat attests control-path against novel representation using multiset hash (MSH) function [9]. Unfortunately, the use of MSH prevents the reconstruction of some execution order of the program.

OAT [24] is remote attestation scheme to attest operation integrity of embedded device. OAT defines two types 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 $H = Hash(H \bigoplus RetAddr)$. OAT called those two measurements attestation blob. During verification, the verifier reconstructs paths from the attestation blob. OAT identifies control-flow violation when the 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 the verifier needs to reconstruct the attestation blob.

Chapter 2

Scope

The main work of this research is to study the offline analysis of the ScaRR's model [26] over a set of C programs. Specifically, we write a tool to extract offline measurement using the ScaRR control-flow model, verify the scalability of the algorithm and study its limitation.

To sum up, the main contributions of this thesis are:

1. Propose an implementation of ScaRR's offline measurement extractor.
2. Verify the scalability of ScaRR's offline measurement algorithm.
3. Study the algorithm limitation.

2.1 Outline

Chapter 3: We discuss the background. We present control-flow attack and remote attestation to attest control-flow attack. We also discuss ScaRR control-flow model. Last, we present LLVM, which we use to extract the control-flow graph (CFG), to mark the basic block as checkpoints and find the list of actions between those checkpoints.

Chapter 4: We present the methodology based on the scope of the contributions. We start by discussing the threat model. We then mainly shows the algorithms and implementation of the offline measurement extractor.

Chapter 5: We show the result of the program implementation and experiment. We demonstrate how to run the LLVM pass. Then, we present the control flow result for some programs that we test. We also discuss the complexity analysis of the implementation. Last, we present one of the programs as a case study. We show the breakdown of the process, from showing the code, get the control-flow graph, and ultimately how we get the offline measurement result.

Chapter 6: We discuss the limitation and future works for the research.

Chapter 7: We close this thesis with conclusion that we gather after the research.

Chapter 3

Background

We present a brief history of memory attacks and some background information on control-flow attacks 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. We present an overview of LLVM that is relevant to the research in Section 3.4.

3.1 Control Flow Attack

A control-flow attack happens when an adversary maliciously makes a program act on his choice. The adversary can do so without statically modifies the program binary but alters the runtime properties of the program. The adversary intention can be to execute malicious operations or to leak secret information. Many of these runtime software security attacks occur due to memory corruption bug in software written in low-level languages like C and C++ [25].

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 [23, 22] and data only attack [8, 7].

We can classify control-flow hijack into code injection attacks and code reuse attacks. Code injection attack injects malicious codes to the program. The malicious code executes action prepared by the attacker. We can mitigate code injection attacks using solutions like non-executable stack (NX), Data Execution Prevention and $W \bigoplus R$ [27]. Code reuse attack executes malicious action without injecting any codes. Return oriented programming [21] is example of code reuse attack. Return-oriented program chains together short instruction sequences already present in the program. Specifically, adversaries use instruction that ends in a `return` opcode. Unfortunately, ROP cannot be mitigated by $W \bigoplus R$ [21].

Memory error attacks and defenses are 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 types of research in this area which we briefly present in Chapter 1.

3.2 Remote Attestation

In this thesis, we explore the use of remote attestation in detecting control-flow attacks. The goal of remote attestation is to validate the state of a remote system. Generally, we implement the remote attestation as protocol with two parties, the verifier as the attester and the remote system as the prover. Prover provides evidence to verifier over a network [10]. The ubiquitous deployment of IoT and different applications in the cloud require a robust remote attestation method. A good remote attestation can ensure the detection of any attacks. Initially, remote attestation only covers static attestation of the application binary. Recently, more sophisticated attacks — that can alter application behaviors — emerges so that static attestation does not suffice.

As we mention above, there are two roles involved in the remote attestation. They are a trusted prover and a verifier. A prover is the one that must prove that the software's runtime integrity. Verifier checks prover to ask the current state of the runtime of the program. Alternatively, the prover also can update the verifier periodically without an explicit trigger from the verifier. The verifier checks the prover response with the local database. If any of measurement mismatches, it means the has been a violation due to the attacks.

This research mainly focuses on offline measurement data generation for remote attestation. The verifier uses the offline measurement in the local database to validate the control-flow graph. We discuss the detail of the control-flow model in Section 3.3. We write the offline program analyzer using LLVM. We discuss some LLVM backgrounds in Section 3.4.

3.3 ScaRR Control-Flow Model

ScaRR [26] takes lesson learned from many former runtime remote attestation scheme to build a model that can perform in a scalable way and can perform remote attestation on a complex system. ScaRR control-flow model consists of two main components: checkpoint and list of action.

As with many previous runtime attestation schemes, ScaRR models and validates the attestation based on the 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 delimits the 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 an 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, some different checkpoints are marked in the program CFG.

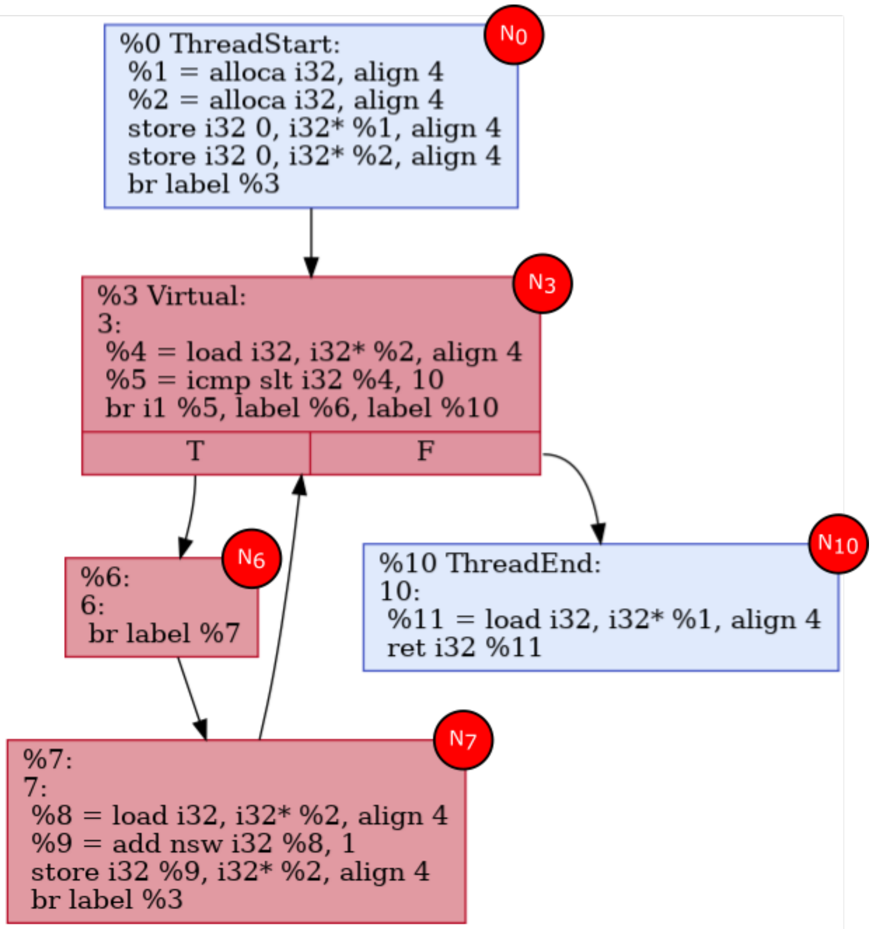
3.3.2 List of Actions

List of actions (LoA) are edges (marked by two checkpoints) that direct one checkpoint to the next one. In a 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 the CFG in the Figure 3.1. The LoA between N_3 (Checkpoint Virtual) and N_{10} (checkpoint ThreadEnd) is $[(BBL_3, BBL_{10})]$. However, the LoA between N_0 and N_3 is [] (empty set).



```
%0 ThreadStart:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 0, i32* %2, align 4
 br label %3
```

```
%3 Virtual:
3:
 %4 = load i32, i32* %2, align 4
 %5 = icmp slt i32 %4, 10
 br i1 %5, label %6, label %10
```

```
%6:
6:
 br label %7
```

```
%10 ThreadEnd:
10:
 %11 = load i32, i32* %1, align 4
 ret i32 %11
```

```
%7:
7:
 %8 = load i32, i32* %2, align 4
 %9 = add nsw i32 %8, 1
 store i32 %9, i32* %2, align 4
 br label %3
```

FIGURE 3.1: Loop CFG.

3.4 LLVM

LLVM is a compiler framework developed by Chris Lattner. LLVM provides portable program representation and different compiler toolings. LLVM supports the implementation of various frontends, backends, and middle optimizers for many programming languages [18].

3.4.1 Intermediate Representation

LLVM intermediate representation (IR) is a high-level representation of programs that enables analysis, instrumentations, and transformations. However, the IR is sufficiently low-level to represent arbitrary programs and to allow broad optimization.

Listing 3.1 shows a simple C program whose the IR of the code is in Listing 3.2. The text representation is just one form of IR. Besides 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. We group instructions into basic blocks: containers for instructions that execute sequentially. This arrangement explicitly represents the application control-flow graph (CFG) in the IR. The details of LLVM IR is available in the Language Reference. [1]

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 applies transformations — which may include some analysis pipelines — and optimizations using tools called `opt`. `opt` takes LLVM IR (either as text, bytecode, or in memory) as input and then does transformations, analysis, and optimizations on it (see Figure 3.3). Transformation and optimization alter the LLVM structure. The analysis gets information from the structure, which usually to be used by one or more transformations. LLVM performs different transformations, optimizations, and analyses as pipelines of passes. LLVM pass can run per function, module or loop. LLVM executes function pass 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, to use the legacy approach. The latest one, to use the new pass manager approach. The approach is different (1) in structuring the code for the pass and also (2) 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.

In the legacy approach, we need to provide the pass name as a 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). LLVM executes the passes in order.

[1] https://llvm.org/docs/LangRef.html

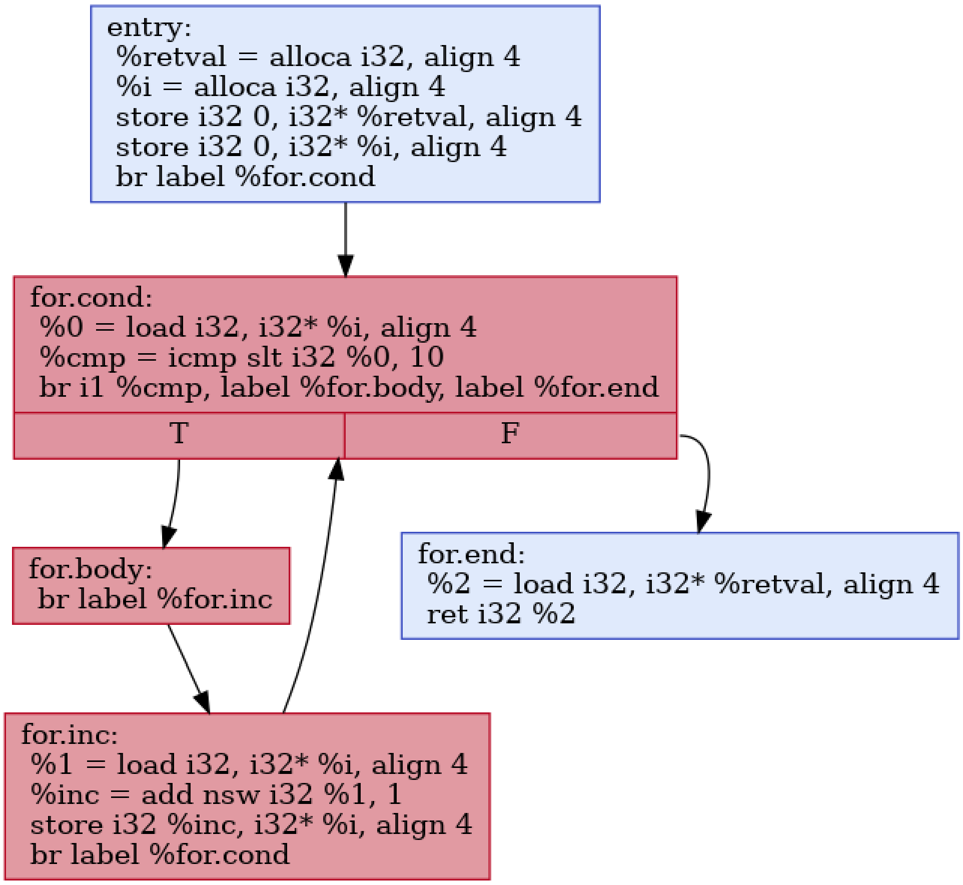

entry:
%retval = alloca i32, align 4
%i = alloca i32, align 4
store i32 0, i32* %retval, align 4
store i32 0, i32* %i, align 4
br label %for.cond
for.cond:
%0 = load i32, i32* %i, align 4
%cmp = icmp slt i32 %0, 10
br i1 %cmp, label %for.body, label %for.end
T
F
for.body:
br label %for.inc
for.end:
%2 = load i32, i32* %retval, align 4
ret i32 %2
for.inc:
%1 = load i32, i32* %i, align 4
%inc = add nsw i32 %1, 1
store i32 %inc, i32* %i, align 4
br label %for.cond

FIGURE 3.2: CFG for Simple C Program.

3.4.3 LLVM API

We use LLVM API to write LLVM pass. In this section, we present relevant required components in implementing LLVM pass for the offline program analysis. LLVM API uses many C++ features and libraries such as Standard Template Library (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 elements of the API is available in the Programmers Manual. [2] We also can refer to the Complete API documentation in the Doxygen page. [3]

Module

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

[2] https://llvm.org/docs/ProgrammersManual.html
[3] https://llvm.org/doxygen/

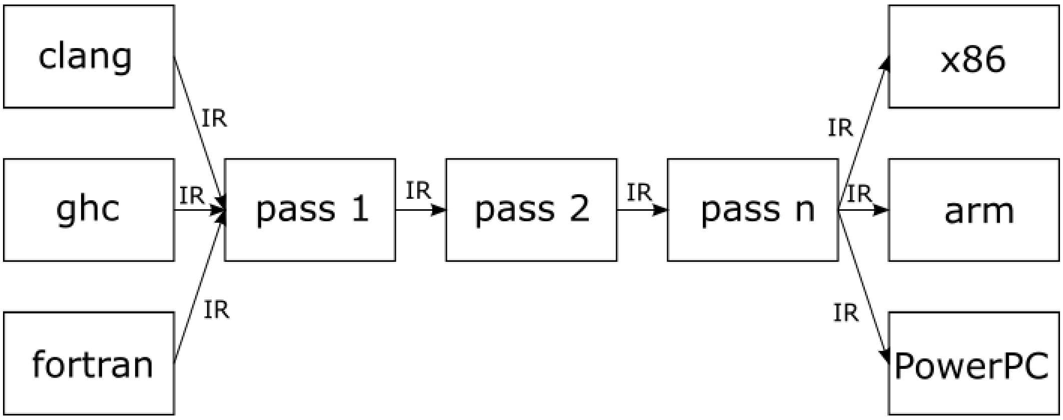


FIGURE 3.3: LLVM Pass.

In LLVM pass, we can get access to a 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 function within module is as simple as iterating the module in a loop. Module provides iterator that returns list of functions in the module (see Listing 3.5).

Function

Function in LLVM represents a function in the source program. A function contains a list of zero or more BasicBlocks. There is one entry BasicBlock and can be multiple exit BasicBlocks. We can get a handler to a function either by getting the iterator from a module instance or by implementing 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 functions before feeding the IR into the ScaRR passes. See also Section 4.3 which discusses the inlining process.

Basic Block

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

Graph Traversal

LLVM CFG has already been structured as graphs. Therefore the basic block can be traversed using different ready-to-use graph traversal algorithms. 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 frontends 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.

We can set 0 (no optimization) to 3 (maximum optimization) to the optimization flag of `clang`. Optimization level 3 makes programs run faster but produces a larger program size.

By default, clang strips out value names and optimizes the code 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.

opt

`opt` is the LLVM optimizer and analyzer. We can invoke `opt` from the command line and tell which LLVM pass that we want to run. We use `opt` to execute the offline program analyzer to marks basic block checkpoints and calculate list of actions. We use the information from LLVM pass via `opt` to detect control-flow violations during remote attestation.

cmake

`cmake` is a build file generator that has an important role in large projects like LLVM. Although we do not require a deep understanding of `cmake` in implementing LLVM pass, we need to know at least how to build the LLVM source code that we write.

We can download LLVM using git (see Listing 3.9). We implement the offline measurement extract using LLVM 13.0.0. After downloading the code, 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.

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

```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.

```
        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.

```
#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.

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

LISTING 3.6: LLVM Basic Block API.

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

LISTING 3.7: Compiling C to LLVM IR.

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

LISTING 3.8: Compiling C to LLVM IR without Optimization.

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

LISTING 3.9: Cloning LLVM Source Code.

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

Chapter 4

Methodology

In this chapter, we present the methodology of the research. First, we present the threat model in section 4.1. We show the overview of ScaRR algorithm to extract checkpoints and list of actions in section 4.2. We discuss code inlining in Section 4.3. Section 4.4 discusses the LLVM implementation to get checkpoints. Section 4.5 discusses the methodology of getting list of actions. We use checkpoint and list of actions to build the offline measurement database for remote attestation.

4.1 Threat Model

We take the threat model in this research from ScaRR [26]. There are two parties: attacker and prover.

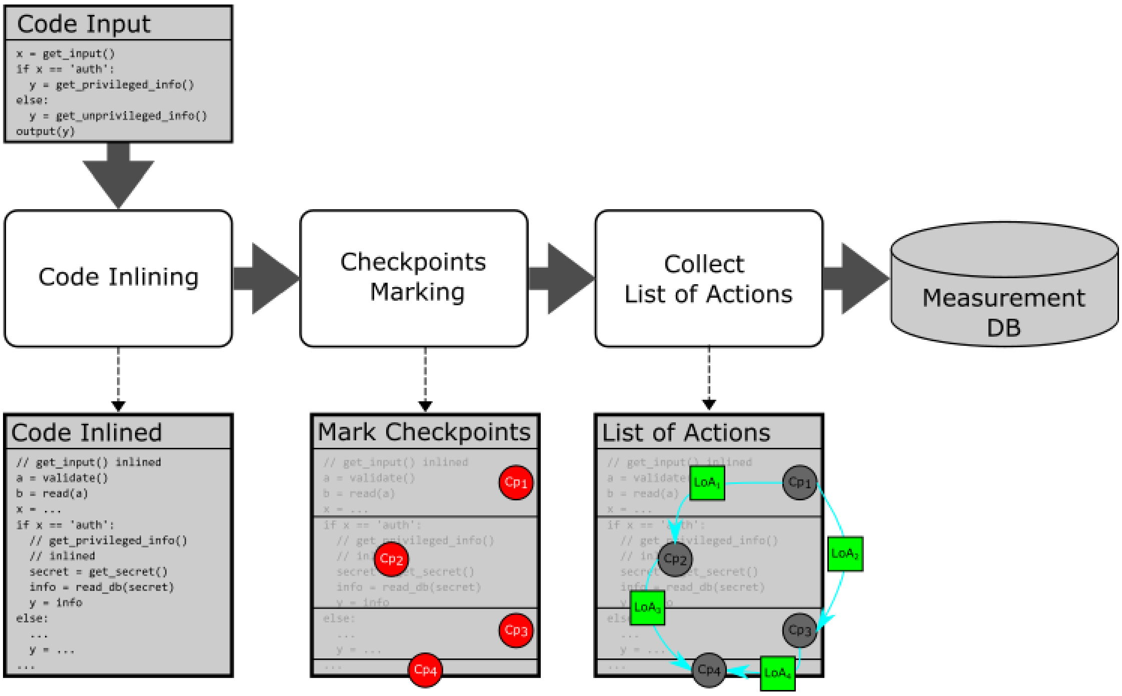
Attacker capabilities: The attacker aims to control remote service using various methods such as memory attacks or any attack in user-space. The attacker has bypassed memory attack protection such as Control Flow Integrity (CFI) or $W \bigoplus R$ or ASLR using techniques like Return-Oriented Programming (ROP) [21] 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, has common memory corruption attack mitigations such as $W \bigoplus R$ and ASLR. Finally, the defender can statically measure the integrity of the Prover's code (*i.e.,* has a hash representation of the prover's code).

4.2 Overview

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 [26] which we present in the Section 3.3.

Figure 4.1 shows the high-level step on generating the offline measurement. First, we inline the source code input (which could be in an intermediate representation form). Section 4.3 describes the inlining process. Next, we mark the checkpoints from inline code, particularly marking in the basic block of the code. Section 4.4 discusses this checkpoint marking step. Last, for each checkpoint, we collect list of actions which



```
x = get_input()
if x == 'auth':
  y = get_privileged_info()
else:
  y = get_unprivileged_info()
output(y)
```

```
// get_input() inlined
a = validate()
b = read(a)
x = ...
if x == 'auth':
  // get_privileged_info()
  // inlined
  secret = get_secret()
  info = read_db(secret)
  y = info
else:
  ...
  y = ...
...
```

FIGURE 4.1: Generating Offline Measurement.

indicates how a process traverses to move from one checkpoint to another. Section 4.5 talks about the process of getting list of actions.

4.3 Code Inlining

One most common compiler optimization is inlining. Inlining extracts a body of a function to the function that calls it. For the context of ScaRR control-flow model, inlining helps to generate correct checkpoints and list of actions. There is a checkpoint called exit-point that require code inlining for accurate identification. Exit-point appears at function call instruction in a basic block whose calls an external function. Inlining the code reduces most inter-translation-unit function calls. Hence it makes identifying the exit-point checkpoints easier. In LLVM, we can use an inlining pass that inlines internal function within the same translation unit.

Consider a program in Listing 4.1. The main function calls another internal function. Without inlining, the `main` function control-flow graph shows no branches and the detail of logic in the factorial function. We cannot generate accurate offline measurements with this original form. However, after we inline, we can see branches in the control-flow graph, which represent the loop in the code. In this inlined representation, we can generate more accurate offline measurements. Figure 4.2 shows the factorial CFG without inlining and with inlining.

```c
#include <stdio.h>

int factorial(int n) {
    int acc = 1;
    while (n > 0) {
        acc *= n;
        n -= 1;
    }
    return acc;
}

int main() {
    printf("factorial(6) = %d\n", factorial(6));
}
```

LISTING 4.1: Program to calculate factorial.

4.4 ScaRR Checkpoint Marker

Offline measurements use checkpoints and list of actions. We represent offline measurement as a triplet. The first and second element is checkpoint; the last element is hashes of the list of actions. The verifier checks the offline measurement database during remote attestation.

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

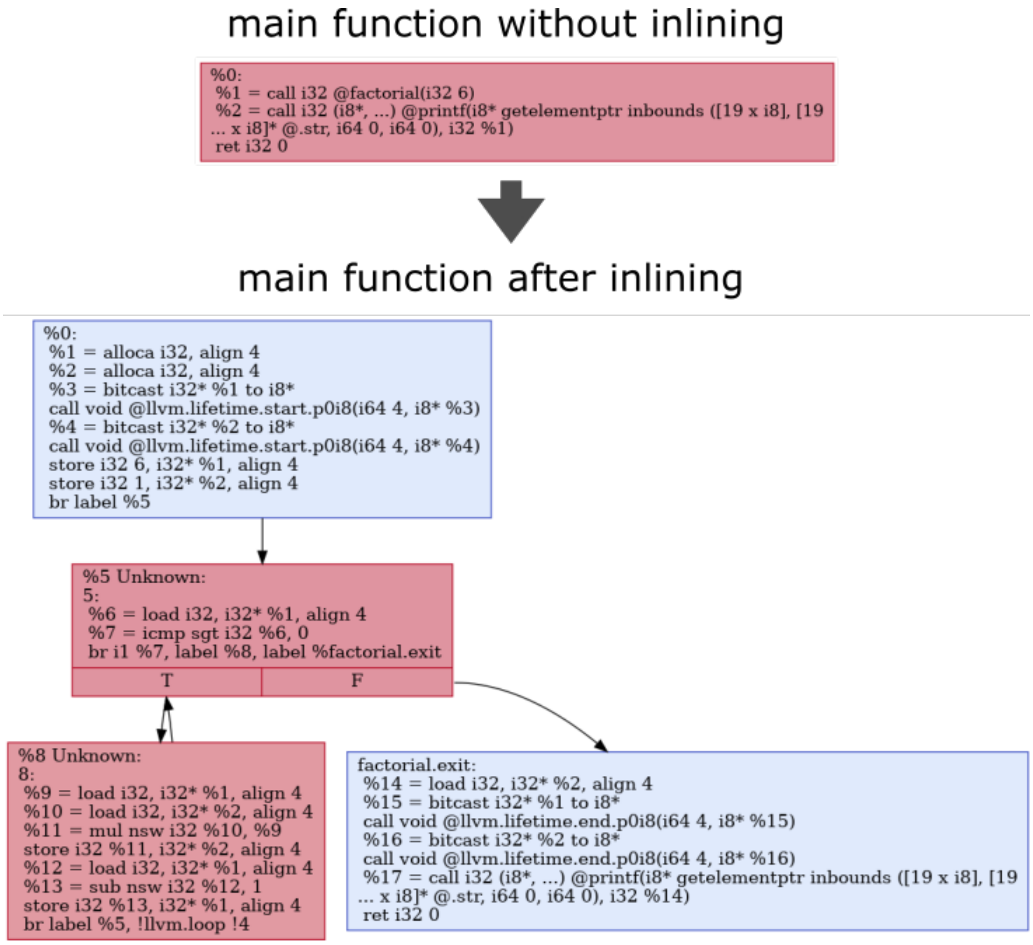
We need program's control-flow graph (CFG) from inlined codes to generate checkpoints. In Chapter 3 we list these four types of checkpoints: **Thread Begin**, **Thread End**, **Exit Point** and **Virtual Checkpoint**. Now we present the heuristic on how we mark it as a checkpoint when appropriate.

The logic of the checkpoint marker is to traverse the control flow graph at least once. We check the basic block checkpoint types as we traverse the basic blocks. We modify the `BasicBlock` class to add checkpoint instance variable as shown in Listing 4.2, so that we can store checkpoint type.

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

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

Exit-point marks that a basic block that calls function outside of translation unit. The



```
%0:
 %1 = call i32 @factorial(i32 6)
 %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([19 x i8], [19
... x i8]* @.str, i64 0, i64 0), i32 %1)
 ret i32 0
```

```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = bitcast i32* %1 to i8*
 call void @llvm.lifetime.start.p0i8(i64 4, i8* %3)
 %4 = bitcast i32* %2 to i8*
 call void @llvm.lifetime.start.p0i8(i64 4, i8* %4)
 store i32 6, i32* %1, align 4
 store i32 1, i32* %2, align 4
 br label %5
```

```
%5 Unknown:
5:
 %6 = load i32, i32* %1, align 4
 %7 = icmp sgt i32 %6, 0
 br i1 %7, label %8, label %factorial.exit
```

```
%8 Unknown:
8:
 %9 = load i32, i32* %1, align 4
 %10 = load i32, i32* %2, align 4
 %11 = mul nsw i32 %10, %9
 store i32 %11, i32* %2, align 4
 %12 = load i32, i32* %1, align 4
 %13 = sub nsw i32 %12, 1
 store i32 %13, i32* %1, align 4
 br label %5, !llvm.loop !4
```

```
factorial.exit:
 %14 = load i32, i32* %2, align 4
 %15 = bitcast i32* %1 to i8*
 call void @llvm.lifetime.end.p0i8(i64 4, i8* %15)
 %16 = bitcast i32* %2 to i8*
 call void @llvm.lifetime.end.p0i8(i64 4, i8* %16)
 %17 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([19 x i8], [19
... x i8]* @.str, i64 0, i64 0), i32 %14)
 ret i32 0
```

FIGURE 4.2: Original and Inlined Factorial Program.

```
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.

heuristic of marking this type of basic block is we iterate all instructions in a basic block. We check if the instruction is a `call` instruction. Then, 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 mark this as an exit point and stop. If none of the instructions is a `call` instruction or all `call` instruction in this basic block call function with nonempty basic block, it means this basic block calls an internal function. Therefore, this basic block is not an exit point. Listing 4.3 shows the snippet to find ExitPoint.

```
        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

```
    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 function
            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

```
function collectLoaRecursive(firstCheckpoint, successorBasicBlock) {
    if (firstCheckpoint has more than one successor and
        no LoA is added yet) {
            add firstCheckpoint as 1st (cpA) LoA for this path
    }

    for (succ = successors of successorBasicBlock) {
        if (succ is a checkpoint) {
            if (one LoA has been added for this path) {
                add succ as the 2nd (cpB) LoA
            }
            add cpA and cpB LoA to measurement
        } else if (succ is not a checkpoint) {
            if (LoA is non empty and the previous LoA is a checkpoint) {
                add succ as the 2nd LoA
            }
            recursively collectLoaRecursive(firstCheckpoint, succ)
        }
    }
}

function collectLoa(list of basicBlock) {
    for (basicBlock = all basicBlocks) {
        collectLoaRecursive(basicBlock, basicBlock)
    }
}
```

LISTING 4.5: Pseudocode to Collect List of Actions.

Virtual checkpoint is a checkpoint that marks special cases such as loop or recursion. We discuss only for loop case in this thesis. Virtual checkpoint in a loop is 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 provides it in LoopInfoBase API. See Listing 4.4

4.5 ScaRR LoA Collector

Checkpoints information is a requirement to get the list of actions. We traverse the paths between two checkpoints and add significant basic blocks that direct the path between the two checkpoints. Those basic blocks comprise the list of actions.

The algorithm of getting LoA between two checkpoints is a little bit more complex. First, we iterate all the basic blocks. Once we find a basic block with some checkpoint types, we mark this as 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 checkpoints, the LoA is an empty set. If there is a branch, the first LoA is always cpA and the second LoA is always be the first basic block after the branch — which can be cpB or just a non-checkpoint basic block. We show the pseudocode for this logic in Listing 4.5. Interested readers can refer to the source code of this pass to see the detail.

Chapter 5

Results

In this research, we use 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. We analyze and present the result in this chapter. The analysis and source code of the program is available online.[1]

We choose four large open-source software projects for analysis. The first software is Redis 6.2.4 [20], a so-called data structure server. The Redis source build consists of the server binary and the client command-line interface (CLI). We analyze both of the programs. The second program we analyze is bzip2 1.0.8 [6]. Bzip2 is a free and open-source file compression program. The third program is OpenSSL 1.1.1j [19]. OpenSSL is a full-featured toolkit for TLS protocol. The last program we analyze is Coreutils 8.32 [11]. Coreutils is a suite of Unix utilities for file, shell, and text manipulation.

5.1 Running LLVM Pass

We write the tool to extract the ScaRR offline measurement using LLVM. Specifically, we implement the tool as LLVM pass. We discuss some background on LLVM pass in Subsection 3.4.2 in Chapter 3. In this section, we show how to use the pass.

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

LISTING 5.1: Mark Checkpoint in BasicBlock.

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

LISTING 5.2: Print Checkpoints in CFG dot file.

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

LISTING 5.3: Get List of Actions.

[1] Our offline measurement generator is available here https://github.com/lamida/scarr-sample-program/.

```
=============================================================
ScaRR Offline Measurement Statistics
=============================================================
Offline Measurement Size: 151
Number of Checkpoints: 94
Number of List of Actions: 216
csv,../cat.ll,119,151,94,216
=============================================================
Checkpoints and LoA Details:
============================================================
... the detail output ...
```

LISTING 5.4: Output of ScaRR Offline Measurement.

We can invoke LLVM `opt` to mark the list of Checkpoints as shown in Listing 5.1. We can see the output of the marked basic block using LLVM `dot-cfg` pass. The commands in Listing 5.2 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 5.3. Note that we have to run `scarr-cp-marker` before `scarr-loa-collector`. Listing 5.4 shows output of the offline measurement extractor.

5.2 ScaRR Control-Flow Result

For each program, we collect the following measurements. The text in parenthesis are the abbreviations that we use for the column headers in Table 5.1. In particular, we measured:

- Source code lines (CL)
- IR lines (IRL)
- Number of basic blocks (BB)
- Number of ScaRR measurements (M)
- Number of checkpoints (CP)
- Number of LoA (LoA)

We can see the trend from the measurement in the table by sorting the data first. Figure 5.1 presents the measurements result by comparing the number of basic blocks against three different measurement metrics: number of checkpoints, number of LoAs and number of measurements. We can see there is a linear relationship among the metrics, which verifies our assumption.

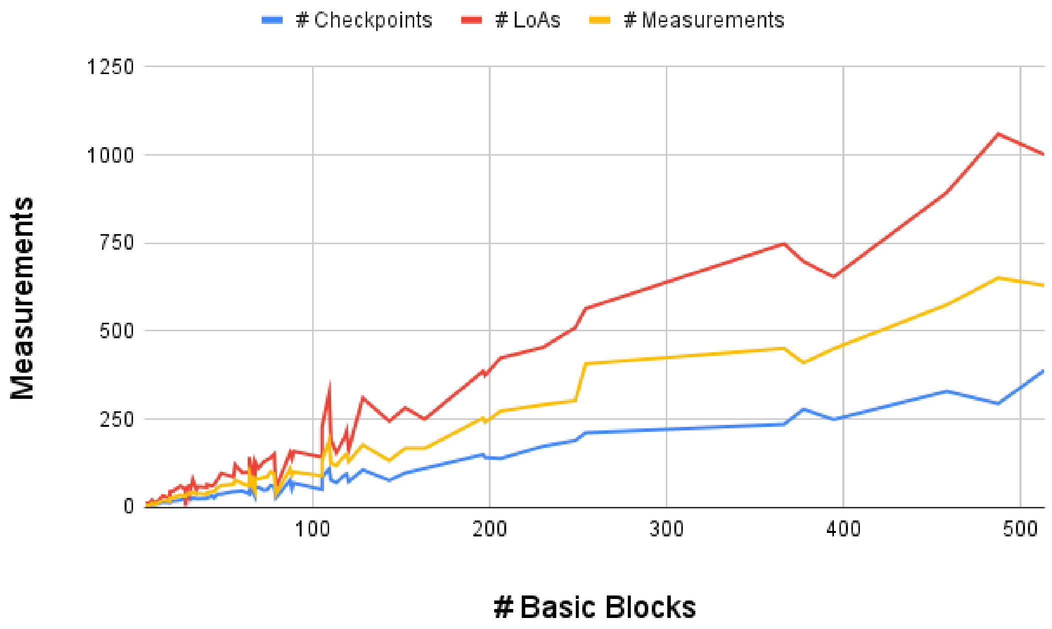


FIGURE 5.1: Measurement Results.

Program	CL	IRL	BB	M	CP	LoA
Coreutils						
basename	190	827	19	17	13	26
cat	767	2215	119	151	94	216
chgrp	319	1158	43	44	32	60
cksum	310	1012	9	11	8	20
cp	1226	3619	79	37	27	56
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
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
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
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
Redis						
redis-benchmark	1982	29367	254	407	211	564
redis-cli	8400	45836	513	630	388	1002
server	6397	25790	88	91	52	138
Bzip2						
bzip2	2036	8542	143	132	76	244
OpenSSL						
openssl	832	1793	28	30	19	42

TABLE 5.1: ScaRR Offline Measurement Result.

5.3 Complexity Analysis

We split the complexity analysis of the program into time complexity and space complexity. The time complexity of Checkpoint Marker is $O(n)$. We only need to traverse the basic block once to identify whether a basic block is a checkpoint. The time complexity of LoA collector is also $O(n)$. We traverse the basic block one time either by breadth-first or depth-first list of the basic block. First, we mark the first checkpoint as the first basic block as we move in traversing. We stop as soon as we find the next checkpoint or a first basic block after branch. We can see the time complexity of LoA collector is $O(n)$.

The space complexity for both algorithms is also $O(n)$. The only additional storage that we require is to store N number basic block during marking the checkpoint and then collecting the LoA.

5.4 Case Study

We present a simple case study on how we process a program and get the offline measurement. We consider a program in Listing 5.5. The IR is in the Listing 5.6. We can see the basic block of the program in Figure 5.2.

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

LISTING 5.5: Case Study Program.

The first step to get the offline measurement is to inline the code in the basic block. In this simple program, we cannot inline the code further, because there is no function call at all.

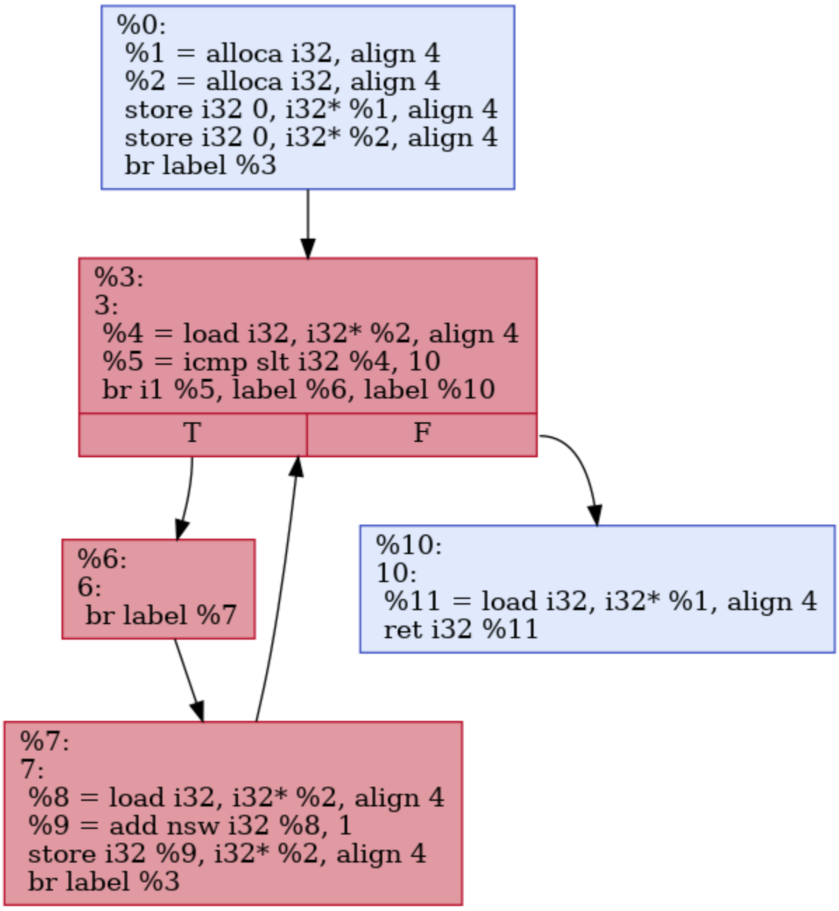

%0:
%1 = alloca i32, align 4
%2 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 0, i32* %2, align 4
br label %3
%3:
3:
%4 = load i32, i32* %2, align 4
%5 = icmp slt i32 %4, 10
br i1 %5, label %6, label %10
T
F
%6:
6:
br label %7
%10:
10:
%11 = load i32, i32* %1, align 4
ret i32 %11
%7:
7:
%8 = load i32, i32* %2, align 4
%9 = add nsw i32 %8, 1
store i32 %9, i32* %2, align 4
br label %3

FIGURE 5.2: Simple Loop Basic Block.

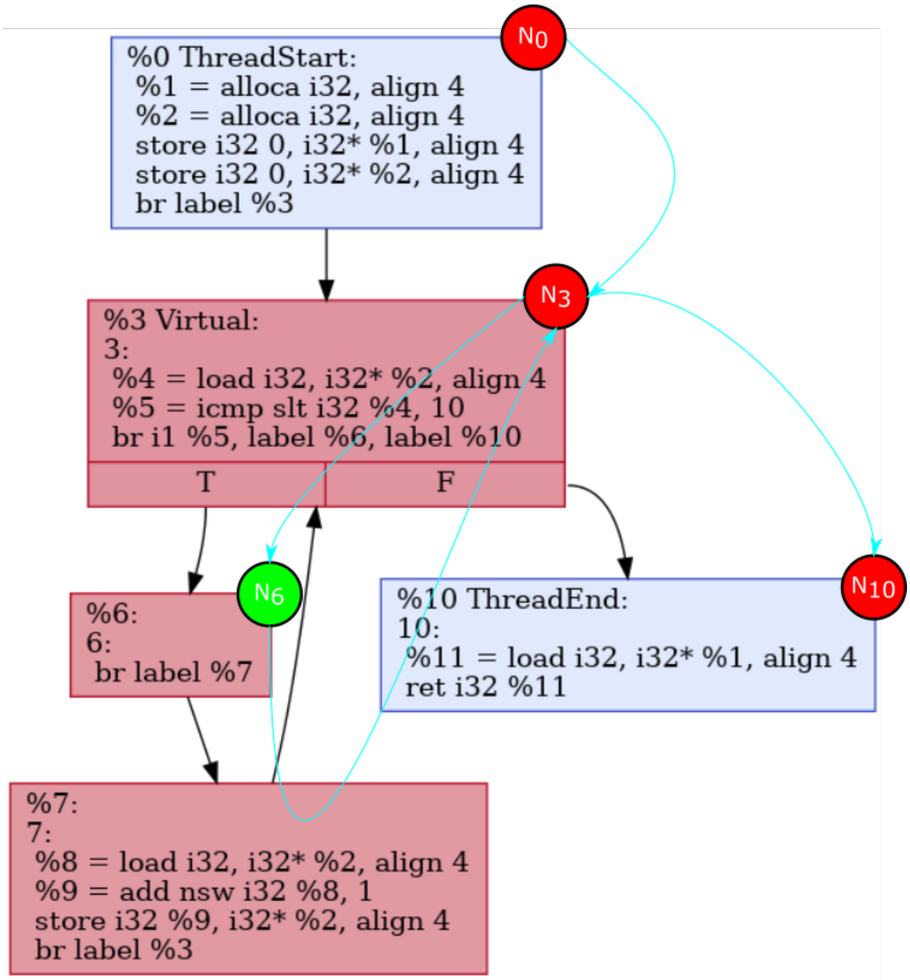
After we inline the code, we traverse the basic block and use heuristic to marks some basic blocks to some checkpoints types. The next step, the measurement extractor traverse each path between two checkpoints and collect the list of action between them. We show the visual of measurement result in Figure 5.3.

The actual measurement between the checkpoints is the following. In summary, this program has 5 lines of source code and around 39 IR source lines. The program has 5 basic blocks, 4 LoAs and 4 measurements.

$$N_0 - N_3 \Rightarrow []$$
$$N_3 - N_3 \Rightarrow [N_3, N_6]$$
$$N_3 - N_{10} \Rightarrow [N_3, N_{10}]$$

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

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  br label %3

3:                                                ; preds = %7, %0
  %4 = load i32, i32* %2, align 4
  %5 = icmp slt i32 %4, 10
  br i1 %5, label %6, label %10

6:                                                ; preds = %3
  br label %7

7:                                                ; preds = %6
  %8 = load i32, i32* %2, align 4
  %9 = add nsw i32 %8, 1
  store i32 %9, i32* %2, align 4
  br label %3

10:                                               ; preds = %3
  %11 = load i32, i32* %1, align 4
  ret i32 %11
}

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

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

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

LISTING 5.6: Case Study Program IR.



```
%0 ThreadStart:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 0, i32* %2, align 4
 br label %3
```

```
%3 Virtual:
3:
 %4 = load i32, i32* %2, align 4
 %5 = icmp slt i32 %4, 10
 br i1 %5, label %6, label %10
```

```
%6:
6:
 br label %7
```

```
%10 ThreadEnd:
10:
 %11 = load i32, i32* %1, align 4
 ret i32 %11
```

```
%7:
7:
 %8 = load i32, i32* %2, align 4
 %9 = add nsw i32 %8, 1
 store i32 %9, i32* %2, align 4
 br label %3
```

FIGURE 5.3: Simple Loop Checkpoints and LoAs.

Chapter 6

Discussion and Future Works

In Chapter 5, we discuss the result of ScaRR's offline measurement generator. We learn that the algorithm time complexity is linear to the input. The input is the size of the program. We measure the size of the program as the size of source code lines. The number of lines is also linear with the size of intermediate representation and basic blocks number. The most important finding is we confirm the number of measurements is also linear with the program size. We know now that we can use the algorithm to do runtime remote attestation for a complex program.

In this study, we identify the limitation of the algorithm. We can use the findings for possible future works. First, we do not test the algorithm with a program that contains recursion, signals, and exception. We also only test the algorithm with a single-thread program. Last, we only run the analysis on C programs.

We also noticed the limitation on the algorithm that cannot detect non-control data attacks. Non-control data attack does not modify program CFG. Next, the algorithm does not track the number of loop iteration. The algorithm does know whether a program is going to a loop. Unfortunately, the algorithm does not know how many iterations a program should run. The offline measurement cannot know when there is an attack that modifies the number of the loop.

Chapter 7

Conclusion

In this thesis, we study the risk of the control-flow attack and how remote attestation can help detect the attack. We implement ScaRR control-flow model extractor. We can use the model to build an offline measurement database for remote attestation to detect control-flow attacks. We present the design and the implementation of LLVM pass to extract the offline measurement. We verify the scalability of the algorithm. We also discuss the limitation of the algorithm and future works.

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] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. en. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. San Jose, CA, USA: IEEE, 2004, pp. 75–86. ISBN: 978-0-7695-2102-2. DOI: 10.1109/CGO.2004.1281665.

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

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

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

[28] 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.