Scalable Runtime Remote Attestation Offline Program Analysis using LLVM

Submitted by

Jon Kartago LAMIDA

Thesis Advisor

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)

July 15, 2021

Declaration of Authorship

I, Jon Kartago LAMIDA, declare that this thesis titled, "Scalable Runtime Remote Attestation Offline Program Analysis using LLVM" 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

Scalable Runtime Remote Attestation Offline Program Analysis using LLVM

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 [18] , 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.

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

List of Abbreviations

LAH **L**ist **A**bbreviations **H**ere
WSF **W**hat (it) **S**tands **F**or

Physical Constants

Speed of Light $c_0 = 2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ (exact)

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

For/Dedicated to/To my...

Chapter 1

Introduction

1.1 Research Motivation

1.2 Research Question

This research are trying to answer these questions.

- How to implement offline program analysis for the ScaRR novel model for remote attestation?
- How is the performance of the analyzer?
- How is the performance comparison against some other attestation scheme such as C-Flat?

Chapter 2

Background

In this chapter, we will present some backgrounds on software attack and remote attestation to detect the attacks. After that, we will share an overview of LLVM that relevant to the research.

2.1 Software Attack

Software attack happens when adversaries make program execution to perform action of their choice. The adversary choice can be executing malicious operations or leaking secret information and more. In the past adversary will modify the binary of the application to statically alter the the behaviour of the program. Nowadays, to prevent detection, attacks are being done by altering runtime properties of the program. 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++ [17].

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 [15, 14] and data only attack [4, 3]. Control-flow hijack can be classified further into code injection attack and code reuse attack such as return oriented programming [13]. Code injection attack is already mitigated by Data Exectuion Prevention. A return-oriented program chains together short instruction sequences already present in a program's address space, each of which ends in a "return" instruction. ROP can't be mititgated by $W \oplus R$ or trusted computing

2.2 Remote Attestation

Remote attestation is the activity of making a claim about properties of a remote target by supplying evidence to an appraiser over a network [6]. The rampant 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 will not suffice.

In remote attestation there will be two roles involved, a trusted prover and a verifier. A prover is the one that must proof that the software hasn't been compromised. Verifier will check prover to ask the current state of runtime of the program. Alternatively, prover also can just update verifier periodically without being asked. The verifier will

compare 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 will be mainly focused on offline measurement data generation for remote attestation which will be used by verifier to validate the control flow graph. We will be using LLVM in implementing the offline program analyzer.

2.3 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 [9].

2.3.1 Intermediate Reprsentation

LLVM intermediate representation (IR) provides high-level information about programs to support sophisticated analysis and transformations but low-level enough to represent arbitrary programs and to allow extensive optimization in it. As an example, consider a simple C program in listing 1.

The IR of the program can be seen in listing 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. From the C code and the IR above, the CFG graph is as follow.

The details of LLVM IR is available in the Language Reference [10].

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.

2.3.2 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. Transformation and optimization will alter the LLVM structure. Analysis will get information from the structure, which usually will 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 function pass will be executed once for every function in the program. LLVM module pass will be executed once for every module. LLVM loop pass will run 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.

```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 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-i64:64-f80:128-n8:1
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 "correctly-rounded-divide-sqrt-fp-

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

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

LISTING 2: LLVM IR The Sample C Program

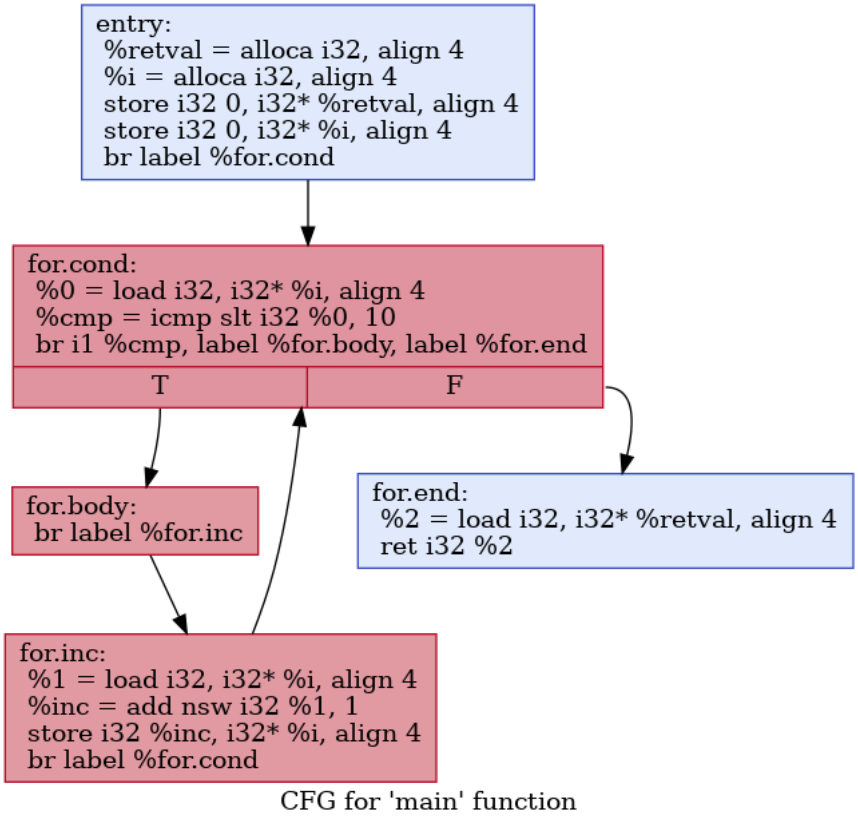

FIGURE 2.1: CFG for Simple C Program

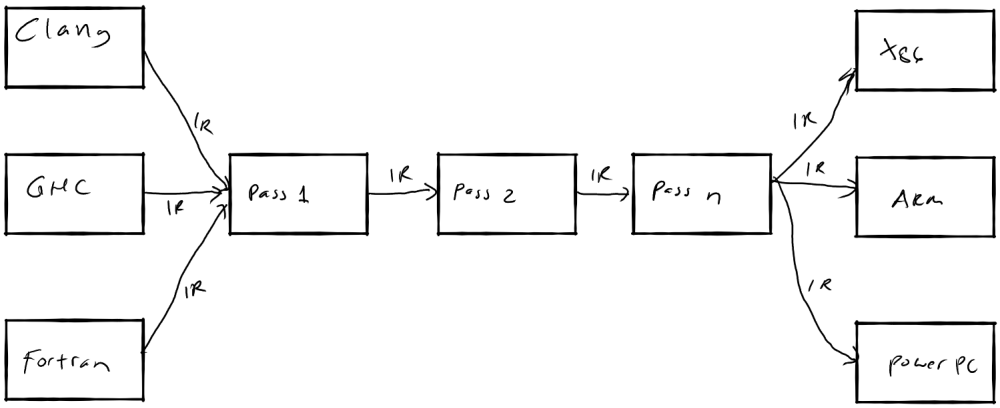

FIGURE 2.2: LLVM Pass (This is image will be replaced with the proper one)

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

LISTING 3: Running Legacy LLVM Pass

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

LISTING 4: Running LLVM New 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 below. In the new pass manager, we are putting the pass name after '–passes' argument in comma separated list. The pass will be executed in order.

2.3.3 LLVM API

In writing LLVM pass, we will use LLVM API. In this section we will 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 [12]. Complete API documentation can be referred at the doxygen page [11].

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 will be as simple as passing module to a loop, since module provides iterator that return list of function in the module.

Function

Function in LLVM represents function in the source program. A function contains list of zero or more BasicBlocks. There will 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.

```
#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 5: LLVM Module API

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

LISTING 6: LLVM Basic Block API

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.

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.

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

LISTING 7: Compiling C to LLVM IR

```
clang -S -emit-llvm -Xclang -disable-O0-optnone -fno-discard-value-names so
```

LISTING 8: Compiling C to LLVM IR without Optimization

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.

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

By default, clang will strip 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.

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 will mark 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. This thesis is implemented on LLVM 13.0.0.

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.

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

LISTING 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
```

LISTING 10: Building LLVM

Chapter 3

Scope of Work

3.1 Implementation

This thesis is implementing the ScaRR Control Flow Model as LLVM passes using LLVM 13.0.0.

3.2 Analysis

We analyzed the control-flow model extracted against different programs with various size and complexities. In this thesis, we are only analyzing program written in C. However, since the LLVM pass is running control from graph extraction against the intermediate representation, we should get consistent result on any programming language that compiles to IR.

Chapter 4

Methodology

In this chapter we will start with brief overview of related remote attestation schemes in the context of generating offline measurement. After that we present the implementation the ScaRR algorithm to extract checkpoint and list of action as LLVM passes. Checkpoints and list of actions are collected to build offline measurement database that is used for the remote attestation.

4.1 Related Attestation Scheme

In this section we present how different attestation scheme encode the offline program representations.

4.1.1 C-Flat (2016)

C-Flat [1] is the first remote attestation scheme to detect runtime control flow attack for embedded systems. C-Flat are generating offline measurement by traversing all possible path of program from start node to the termination node. In each node, C-Flat will hash the node ID and the hash of previous node. In the first node, since there is no previous hash, we will pass 0. This will create hash chains which will be stored as offline measurement database.

4.1.2 Lo-Fat 2017

Lo-Fat[8] is improving C-Flat by using hardware support for control flow attestation. Lo-Fat offline program analysis is still inheriting C-Flat approach.

4.1.3 Atrium (2017)

Atrium [19] 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

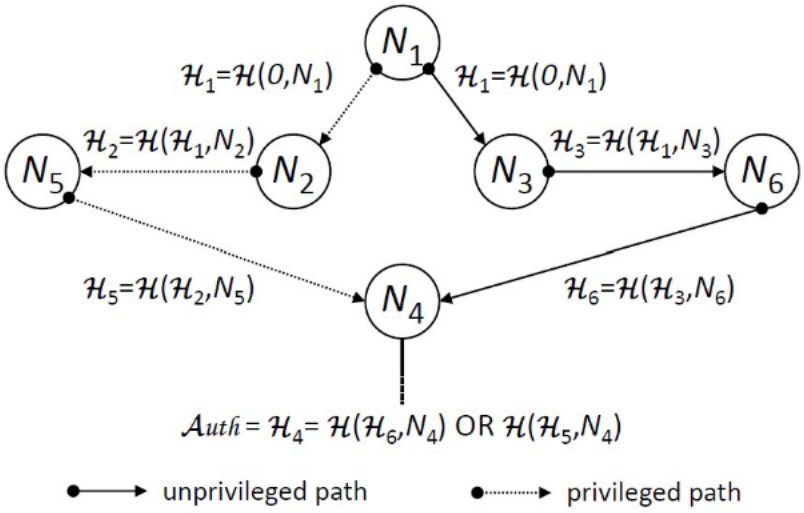

FIGURE 4.1: C-Flat

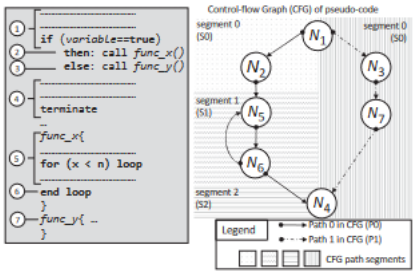

FIGURE 4.2: Atrium

mitigate the TOCTOU attack, the offline measurement generation will still grow exponentially as the complexity of the program grow.

4.1.4 LiteHax (2018)

LiteHax [7] 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 will run symbolic execution and incremental forward data-flow analysis without doing any lookup to offline measurement database.

4.1.5 Diat (2019)

Diat [2] is remote attestation scheme that can attest data integrity and control-flow of autonomous 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 [5]. The use of MSH will make some execution order of the program lost.

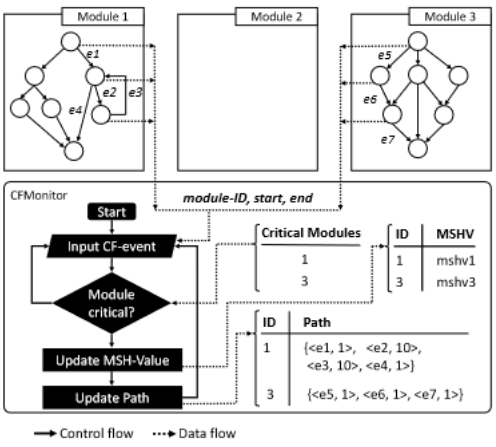

FIGURE 4.3: Diat

4.1.6 OAT (2019)

OAT [16] 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.

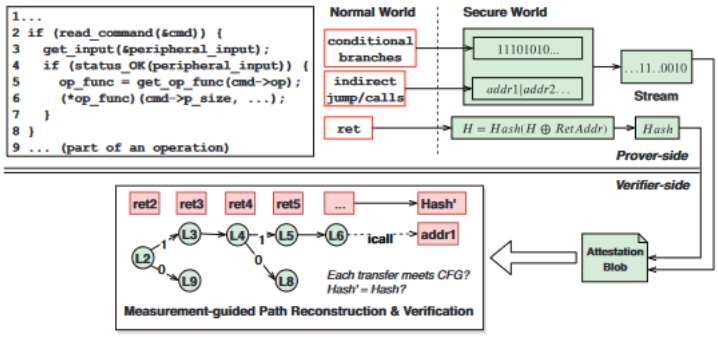

FIGURE 4.4: OAT

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

Although OAT does not encounter the combinatorial hash explosion in C-Flat, there will be verification overhead since verifier needs to reconstruct the attestation blob. TODO compare the overhead with ScaRR.

4.2 ScaRR Control-Flow Model

ScaRR [18] 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.

4.2.1 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 will be marked in the program CFG.

4.2.2 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 the CFG in the Figure 4.5. 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).

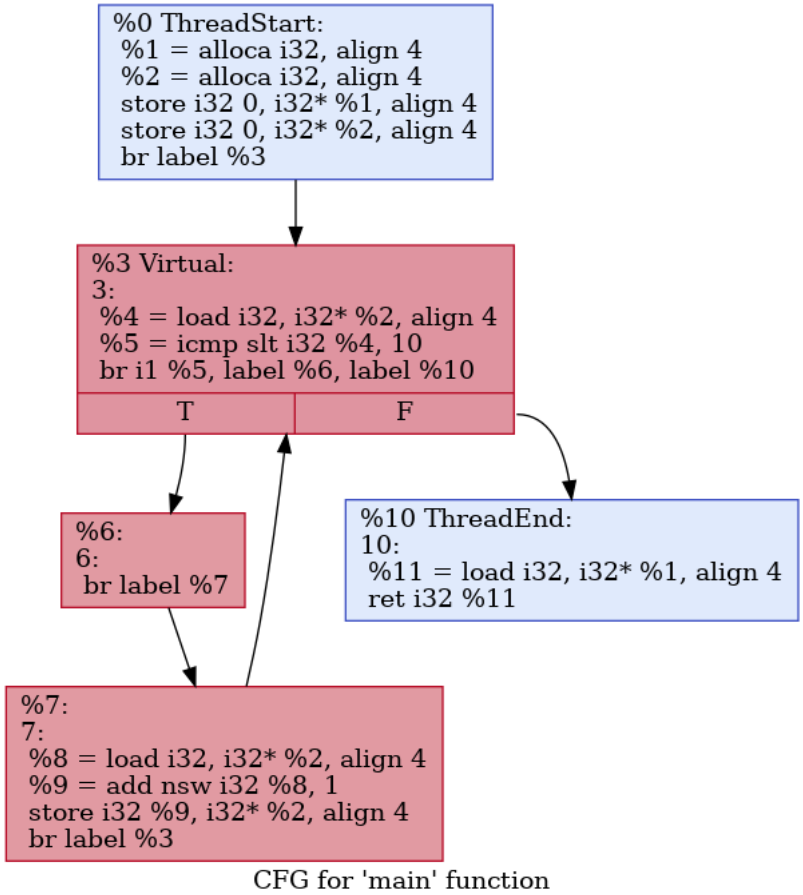

FIGURE 4.5: Loop CFG

4.3 ScaRR LLVM Pass

ScaRR offline measurement is represented as the following key-value pair.

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

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

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

LISTING 11: Add Checkpoint Instance Variable to BasicBlock class

As described in the above, checkpoint is a special basic block that delimit path between execution path in a CFG. cp_A is the start delimiter of the path. cp_B is the end of delimiter. LoA — list of action — is list of significant basic block pairs which define edges

ScaRR extractor will traverse the graph in two passes. The first pass is find whether the basic block is a checkpoint. The second pass will traverse the graph and mark List of Action between every two checkpoint.

4.3.1 ScaRR Checkpoint Marker

The logic of checkpoint marker is to traverse the whole control flow graph at least one and for each basic block, we will 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.

The algorithm to mark the checkpoint is first we iterate all basic block in main function. First we will mark the first basic block with no predecessor as ThreadStart checkpoint and basic block with no successor and ThreadEnd checkpoint.

To identify ExitPoint checkpoint, for each basic block then we iterate each instruction to find whether any instruction in the basic block is a 'call' instruction and has no body defined in this translation unit. Please refer to Listing 12

To mark Virtual checkpoint, we need to find loop header basic block. Although there is no direct API that will tell whether a basic block is a loop header, LLVM provide it in LoopInfoBase API. See Listing 13

4.3.2 ScaRR LoA Collector

The algorithm of getting LoA between two checkpoints are little bit more complex. First we iterate all the basic block and if the basic block is a checkpoint we will mark this is cpA. Next, we will 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 will be an empty set. If there is a branch, the first LoA will be

```
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 12: 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,
    KLoop->releaseMemory();
    KLoop->analyze(DT);
    for (auto &bb : F) {
        // Since the BasicBlock would have been inlined, just traverse f
        if (F.getName() == "main") {
        auto loop = KLoop->getLoopFor(&bb);
        if (loop != nullptr) {
            // found VirtualCheckpoint
                    loop->getHeader()->setCheckpoint(Checkpoint::Virtual
        }
        }
    }
}
```

LISTING 13: Getting Virtual Checkpoint

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

LISTING 14: Mark Checkpoint in BasicBlock

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

LISTING 15: Print Checkpoints in CFG dot file

always cpA and the second LoA will be the first basic block after the branch which can be cpB or just non checkpoint basic block. Interested reader can refer to the implementation of this pass to see the detail.

4.3.3 Running The Pass

To mark the list of Checkpoints, we can invoke LLVM opt as follow.

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

The commands above will generate 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 16

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 16: Get List of Actions

Chapter 5

Results

In this research we used the offline measurement generator in getting the measurement across different programs. We calculated the ScaRR control flow information from each of the program.

5.1 ScaRR Control Flow Result

5.2 Complexity Analysis

5.3 Case Study

Chapter 6

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.

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

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

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

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] 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.