

Compiler Support for Constant-Time Programs in LLVM

by

Mehdi Aghakishiyev

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Mehdi Aghakishiyev 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Side-channel attacks aim to extract sensitive information by monitoring the additional information generated during program execution, such as execution time or power consumption. Certain coding patterns, such as using secret data in control flow and memory addressing instructions, cause the execution time of the program to vary based on secret input, making the program vulnerable to timing-based side-channel attacks. Constant-time programming offers a defense against such attacks; however, it is difficult to implement manually as it requires tracking secret data through complex program logic.

In this thesis, we propose an automated approach to generate constant-time programs based on static analysis and program transformations. First, we use taint tracking to monitor the flow of secret input through the program and mark branching and memory addressing instructions that depend on secret data. Then, we apply program transformation techniques such as branch linearization to remove these dependencies and produce constant-time code. We perform our analysis and transformations on LLVM IR and implement our tool as part of the LLVM Pass Infrastructure.

To evaluate our tool’s effectiveness, we apply our analysis and transformations to programs from the OISA benchmark. We validate our results through BliMe, an architecture performing hardware-enforced taint tracking to prevent side-channel attacks.

Acknowledgements

I would like to thank my advisors, N. Asokan and Meng Xu, for their guidance and support, and my thesis readers: Yousra Aafer and Sihang Liu.

Table of Contents

Author’s Declaration	ii
Abstract	iii
Acknowledgements	iv
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Background	3
2.1 Side-channel attacks	3
2.1.1 Secret dependent branching	4
2.1.2 Secret dependent memory access	5
2.2 Mitigation Strategies	6
2.2.1 Constant-time programming	7
2.2.2 Software-based automated mitigation techniques	7
2.2.3 Architectural-level mitigation techniques	8
2.2.4 BliMe	8
2.3 LLVM	9

3	Problem Description	11
3.1	Problem statement	11
3.2	Requirements	12
3.2.1	Analysis requirements	12
3.2.2	Transformation requirements	13
4	Design	14
4.1	Overview	14
4.2	Analysis	16
4.2.1	Analysis Workflow	16
4.2.2	Abstract Domain	17
4.2.3	Abstract Semantics	20
4.3	Transformations	25
4.3.1	Select Instruction	25
4.3.2	Branch Instruction	25
4.3.3	Load and Store Instructions	34
5	Implementation	36
5.1	Analysis	36
5.1.1	Introducing taint	36
5.1.2	Abstract Domains	37
5.1.3	Analysis Workflow	38
5.2	Transformation	39
5.2.1	Select Transformation	40
5.2.2	Branch Transformations	40
5.2.3	Load and Store Instructions	42

6	Evaluation	43
6.1	Analysis results	43
6.1.1	R-1 Soundness	44
6.1.2	R-2 Precision	45
6.2	Transformation results	45
6.2.1	R-3 Correctness-1: Removing non-constant time code	46
6.2.2	R-4 Correctness-2: Preserving program semantics	46
6.3	Performance evaluation	47
7	Discussion	49
7.1	Limitations	49
7.1.1	Load and Store Transformations	49
7.1.2	Loop execution counts	50
7.1.3	Pre-compiled libraries	50
8	Related works	51
	References	53

List of Figures

4.1 Tool workflow	15
-----------------------------	----

List of Tables

6.1	Number of distinct violations reported	44
6.2	Number of distinct violations reported by Analysis and BliMe after transformations	46
6.3	Time taken by analysis and transformation passes	47
6.4	Execution time of programs in seconds	47

Chapter 1

Introduction

Side-channel attacks rely on observing the program’s execution rather than its output to leak sensitive information. In software side-channel attacks, certain coding patterns—such as using secret data in conditional statements or memory accesses—can cause execution time variations or leak information through observable cache states. Timing and cache side-channel attacks have primarily targeted cryptographic libraries to extract secret keys but have also been observed in other domains, such as compression software [23]. With increased adoption of cloud computing, where resources are shared between multiple users, the attack surface for timing-based side-channel vulnerabilities has grown, increasing their prevalence and impact.

One way of preventing timing-based software side-channel attacks is constant-time programming, where the execution time of the program is independent of the input. For example, secret-data-dependent conditional statements can be linearized to ensure that the same set of instructions is executed in every program instance. However, implementing constant-time programs is difficult as they require the developer to track which parts of the program the secret data is flowing into through complex program logic. A recent survey of developers from 27 cryptographic libraries [11] found that only 13 claimed full resistance and 10 claimed partial resistance against timing attacks. These findings highlight the need for automated techniques to detect and mitigate side-channel vulnerabilities through program transformations.

Several approaches have used automated program transformations to generate semantically equivalent constant-time programs from side-channel vulnerable code. Constantine [5] employs dynamic analysis to track sensitive (tainted) data throughout the program and then applies control flow and data flow linearization techniques to produce a side-channel-

resistant program. However, due to its reliance on dynamic analysis, Constantine cannot cover all execution paths and can potentially miss side-channel vulnerable code. On the other hand, SC-Eliminator [29] and LIF [26] use static analysis to discover side-channel vulnerable code and apply relevant program transformations to eliminate side-channel attacks. However, SC-Eliminator uses intraprocedural analysis, requiring each function parameter to be marked secret or not, and LIF considers a weaker threat model and only addresses memory access instructions affected by secret-dependent branches.

In this work, our aim is to develop a tool to automatically detect timing-based side-channel vulnerable code and apply program transformations to mitigate timing side-channel attacks. In our work, we target branch instructions and memory access instructions that rely on secret data. To track the secret data and its propagation through the program, we rely on static analysis to make sure we cover all program paths and reduce the possibility of missing vulnerable code. We require the developer to provide annotations to mark which values in the program are considered secret and perform taint tracking with marked data being our sources. Our objective is to develop a sound static analysis to find all code regions that are susceptible to timing-based side-channel attacks. As we will apply program transformations to generate constant-time programs based on the analysis results, we also aim for our analysis to have as few false positives as possible because control flow and data flow linearization transformations will incur overhead on the original program. We develop our analysis and transformation tool as part of the LLVM Pass infrastructure, and our transformations are applied to the LLVM IR.

To verify the effectiveness of our tool, we rely on the BliMe [9], an architecture which performs hardware-based taint tracking and causes program aborts if sensitive data flows into observable states, which also include branching and memory addressing instructions. Our goal is for programs compiled with our analysis and transformation pass to execute on the BliMe architecture without any violations.

Our main contributions are as follows:

- A sound static analyzer to detect timing-based side-channel vulnerable code, section 4.2.
- Program transformations to eliminate secret data-dependent branching and memory access instructions to generate constant-time code, section 4.3.
- We plan to make both our tool and the evaluation benchmarks publicly available.

Chapter 2

Background

2.1 Side-channel attacks

Side-channel attacks aim to uncover secret data by observing the execution characteristics of a target program. These characteristics include execution time, memory access patterns, power consumption, electromagnetic radiation, or even acoustic signals emitted during computation. Side-channel attacks have existed for decades and remain relevant today, with new vulnerabilities discovered regularly, such as the SLAP [13] and FLOP [12] attacks on Apple CPUs or the KyberSlash [4] attack on the implementation of the post-quantum cryptography algorithm Kyber Key Encapsulation Mechanism.

This work focuses on timing-based side-channel attacks, which rely on observing the execution time or the memory accesses made by the victim program. In 1996, Paul Kocher theorized timing side-channel attacks on several cryptographic algorithms [14], and in 2003, David Brumley and Dan Boneh [8] demonstrated the practicality of remote timing side-channel attacks by extracting private keys from an OpenSSL-based web server in the local network. Since then, new timing side-channel attacks have been discovered across several domains, such as cryptography libraries [3] [1], compression algorithms [23] [18] or in web applications [24].

The following coding patterns can leave the program vulnerable to timing-based side-channel attacks: (1) secret-dependent branching, (2) secret-dependent memory access, (3) variable-time arithmetic. In this work, we will be targeting the first two cases, as the third case depends highly on the underlying architectural implementations of arithmetic operations and is not as prevalent as the first two. Using secret data in conditional branches

can cause the program to execute different instructions depending on the input, which can lead to execution time differences between different inputs. Similarly, when secret data is used in memory access instructions, the program will exhibit observable time differences in its execution depending on whether the requested data is in the cache or not.

2.1.1 Secret dependent branching

When secret data is used as a branch condition, the program may execute different instructions depending on the secret, which can lead to execution time differences or different memory access patterns. An attacker can repeat this process multiple times to track the variations in execution and reveal the secret information. Listing 2.1 taken from the `libgcrypto` library, shows a side-channel vulnerability discovered in 2013 [30], as a result of an attack known as Flush+Reload, which works by monitoring the access to the cache lines. Here, variable `e` is derived from the secret data, and based on its value, the program will execute additional memory access operations, leaving the program vulnerable to Flush+Reload attack.

```

1  ....
2  if ( (mpi_limb_signed_t)e < 0 )
3  {
4      /*mpih_mul( xp, rp, rsize, bp, bsize );*/
5      if( bsize < KARATSUBA_THRESHOLD )
6          _gcry_mpih_mul ( xp, rp, rsize, bp, bsize );
7      else
8          _gcry_mpih_mul_karatsuba_case (xp, rp, rsize, bp, bsize,
9                                         &karactx);
10
11     xsize = rsize + bsize;
12     if ( xsize > msize )
13     {
14         _gcry_mpih_divrem(xp + msize, 0, xp, xsize, mp, msize);
15         xsize = msize;
16     }
17
18     tp = rp; rp = xp; xp = tp;
19     rsize = xsize;
20 }
21 e <= 1;
22 c--

```

Listing 2.1: libgcrypto secret dependent branching

Listing 2.2 shows the fix that eliminated this vulnerability. In the fixed version, the branch is executed regardless of the value of secret data (variable `e`, and the variable `esec` is used check whether the data is coming from secret source), however, the results are propagated only if the condition on the secret data is met. Although line 18 contains secret dependent branching, operations inside the if block do not perform memory accesses, thus making the program resistant against the Flush+Reload attack.

```

1  ....
2  if (esec || (mpi_limb_signed_t)e < 0)
3  {
4      /*mpih_mul( xp, rp, rsize, bp, bsize );*/
5      if( bsize < KARATSUBA_THRESHOLD )
6          _gcry_mpih_mul ( xp, rp, rsize, bp, bsize );
7      else
8          _gcry_mpih_mul_karatsuba_case (xp, rp, rsize, bp, bsize,
9                                          &karactx);
10
11     xsize = rsize + bsize;
12     if ( xsize > msize )
13     {
14         _gcry_mpih_divrem(xp + msize, 0, xp, xsize, mp, msize);
15         xsize = msize;
16     }
17 }
18 if ( (mpi_limb_signed_t)e < 0 )
19 {
20     tp = rp; rp = xp; xp = tp;
21     rsize = xsize;
22 }
23 e <<= 1;
24 c--;

```

Listing 2.2: libgcrypt secret dependent branching fix

2.1.2 Secret dependent memory access

When a program uses secret data to access memory locations, an attacker can infer information about the secret data by observing the cache access patterns. In Prime+Probe attacks [20], the attacker loads the cache with its own data and waits for the victim program to run. Later, the attacker tries to access its own data again and measures the access time. If the access is slow, this means that the attacker’s data was evicted from the cache by the victim program, letting the attacker know which addresses have been accessed.

Listing 2.3 shows an example where the secret data, the variable `e0`, is used in an indexing operation.

```
1  ....
2  base_u = precomp[e0];
3  base_u_size = precomp_size[e0];
```

Listing 2.3: libgcrypt secret dependent memory access

The above vulnerability was fixed by accessing all possible indexes, but only assigning the correct value through the use of bitwise arithmetics, which can be seen in Listing 2.4. This ensures that the memory access pattern of the program will not depend on the secret data.

```
1  ....
2  base_u_size = 0;
3  for (k = 0; k < (1<< (W - 1)); k++)
4  {
5      struct gcry_mpi w, u;
6      w.allocated = w.nlimbs = precomp_size[k];
7      u.allocated = u.nlimbs = precomp_size[k];
8      w.sign = u.sign = 0;
9      w.flags = u.flags = 0;
10     w.d = base_u;
11     u.d = precomp[k];
12
13     mpi_set_cond (&w, &u, k == e0);
14     base_u_size |= (precomp_size[k] & ((mpi_size_t)0 - (k == e0)) );
15 }
```

Listing 2.4: libgcrypt secret dependent memory access fix

2.2 Mitigation Strategies

As discussed above, side-channel attacks have been around for a long time, and new attacks are being discovered every year. Thus, there have been several approaches proposed to mitigate them, both hardware and software-based. These approaches try to limit the amount of information that can be leaked through the program using several techniques.

2.2.1 Constant-time programming

One way of preventing timing-based side-channel attacks is through constant-time programming. In a constant-time program, the goal is to ensure that the execution time and memory footprint of the program do not depend on secret values, although the overall execution time may vary across executions. In our work, we aim to receive vulnerable version of the program and generate a constant-time equivalent. We achieve this by linearizing secret-dependent branches, such that regardless of the secret value, the same sequence of instructions is executed. Similarly, for memory-accessing instructions, for every secret-dependent memory access, we try to access all possible memory addresses at that point to ensure that the same cache lines are accessed regardless of the secret value.

2.2.2 Software-based automated mitigation techniques

Several tools have been proposed to automatically transform the target program into a constant-time program, which are discussed below.

SC-Eliminator [29] performs static analysis to detect branching and memory addressing instructions that depend on sensitive data. Then, it transforms the secret-dependent branches into linearized control flow. For sensitive memory accesses, SC-Eliminator tries to touch every relevant cache line to prevent cache-timing leaks. However, in their approach, they consider a less-capable attacker, who can only observe the total execution time of the program, and do not consider attackers that can observe latency in cache accesses as described in Prime+Probe attacks. Similar to SC-Eliminator, LIF [25] also relies on static analysis for the detection of side-channel vulnerable code, specifically secret-dependent branches. However, memory accesses affected by secret input are not being considered, leaving the problem susceptible to cache-based timing attacks.

Constantine [5] relies on dynamic analysis to detect secret-dependent code and also to infer loop execution counts, which will be used in later transformations. After obtaining analysis results, they apply control flow and data flow linearization techniques to mitigate timing-based side-channel vulnerabilities. Similar to Constantine, Pendulum [22] also utilizes dynamic analysis to detect side-channel vulnerable code. They use fuzzing to identify code regions that experience timing variation depending on the secret input. Then, pattern-based code repair techniques are applied to minimize the timing variation to prevent side-channel attacks. However, because both tools rely on dynamic analysis, it is possible that they can only detect side-channel vulnerable code that is in their execution path, and may miss potentially vulnerable code regions. Furthermore, patterns included

in Pendulum for code repairs do not include cases for secret-dependent memory addressing instructions.

2.2.3 Architectural-level mitigation techniques

Architectural-level defenses have also been proposed to defend against timing-based side-channel attacks. In ScatterCache [28], the authors propose using a per-process secret key to map the addresses to a cache set. In addition, each address can map to multiple cache sets depending on the cache way, further complicating the attacker’s ability to mount cache timing attacks. CEASER [21] uses a similar approach, where a secret encryption key is used to generate cache set indices, and this key is periodically updated, changing the mapping between physical address and encrypted address, invalidating any previous information that the attacker has gathered on the victim’s code. In another work [27], the authors modified the processor frequency, which affected the access times observed by the attacker. They also added random prefetch operations to pollute the attacker’s observations.

While the software-based methods try to eliminate timing-based side-channel vulnerabilities, architectural techniques aim to make it infeasible to exploit potential vulnerabilities.

2.2.4 BliMe

BliMe [9] is another hardware-based approach that prevents data leakage through side-channel attacks. It is an architecture designed to address the problem of client data confidentiality in an outsourced computing environment. Unlike Fully Homomorphic Encryption (FHE), which incurs significant computational overhead, and Trusted Execution Environments (TEEs), which are still susceptible to runtime and side-channel attacks, BliMe achieves its goal by providing a minimal set of extensions to the instruction set architecture that prevents the data from flowing into observable states.

In BliMe, the client data that is not encrypted is referred to as **blinded** data. The client performs a handshake with BliMe hardware and agrees on a session key. Then, the client encrypts its data using the session key and sends it to the BliMe hardware. BliMe atomically decrypts the data and marks it as **blinded**. BliMe enforces a hardware-based taint-tracking policy to monitor the flow of **blinded** data into observable states. The observable states include branches that depend on **blinded** data, load/store instructions using **blinded** data as an address, or instructions whose execution time depends on **blinded** data. When the **blinded** data reaches these states, BliMe triggers a fault and

halts execution. In this way, BliMe prevents timing side-channel attacks, including cache-based attacks, and transient execution attacks as it prevents the `blinded` data to be used in speculative decisions. After the computation is over, BliMe atomically encrypts the data and marks it as `non-blinded` and sends it back to the client.

In our work, we use BliMe as a test oracle to measure the effectiveness of our analysis and program transformations. Our goal is to detect every instance that BliMe reports as a violation and apply program transformations to them. In our analysis, we will potentially find instances that BliMe does not report due to two reasons: (1) program execution never reaches them, thus BliMe does not detect them, (2) false positives resulting from over-approximation in the analysis. Therefore, a program that passes our analysis with no violations should run on BliMe hardware without triggering violations. Furthermore, a program that faults on BliMe should execute safely after program transformations are applied.

2.3 LLVM

LLVM is a modular framework that provides tools and technologies for building compilers. It supports compilation for several high-level languages, such as C/C++, Rust, Swift, and Julia. LLVM works in three phases:

- Frontend, which translates the high-level language to the LLVM Intermediate Representation (IR).
- Middle end, which applies various optimizations to the generated IR.
- Backend, which translates the IR to executable machine code.

The modularity of LLVM makes it easy to add or manage the optimizations applied to the IR, making it a popular tool for developing static analyzers [5][29].

LLVM provides a pass management framework where developers can write custom passes that will be applied to the IR. These passes are divided into two categories: analysis passes that are used to obtain information on the provided program, and transformation passes that modify the IR and produce a new IR. In this project, we utilise both types of passes. First, we implement an analysis pass that receives the input program with developer annotations. A developer can use a custom attribute called `blinded` to mark the sensitive data. This pass performs taint analysis to track the flow of sensitive data and

mark instructions that should be transformed. Later, a transformation pass will receive the analysis results, perform relevant program transformations, and produce a safe IR. By implementing our analysis and transformation as LLVM passes, they can be utilized during the optimization phase, allowing them to be integrated into the build process with appropriate flags to produce an executable that is side-channel-resistant.

Chapter 3

Problem Description

3.1 Problem statement

Through timing-based side-channel attacks, attackers can learn information about users' private data by monitoring the execution time or the memory access patterns of the program. Manually tracking the private data through the program and modifying the code to be side-channel-resistant is difficult, as evidenced by numerous CVEs reported in 2024 in popular cryptographic libraries [1][2][3]. In this work, we aim to detect potential side-channel vulnerable code and apply program transformations to produce a constant-time program.

In our work, we assume that the attacker can observe the execution time of the program, can run their program alongside the victim program, and can infer memory access patterns through cache latency. Furthermore, the attacker is aware of the source code of the victim program and can provide public inputs to the program to observe its execution trace.

In the analysis, our goal is to detect all the program points vulnerable to side-channel attacks while keeping a low false-positive rate to reduce the overhead incurred by the program transformations. As we will be doing our analysis and transformations on LLVM IR, we target the following LLVM instructions:

- Conditional branches
 - `branch`
 - `select`

- Memory instructions
 - `load`
 - `store`

Before the analysis, we assume that the developer annotates all secret inputs (e.g., cryptographic keys). During the analysis, we will treat the annotated inputs as **tainted**, and track their flow throughout the program. Any values that are derived from the tainted data will also be treated as tainted data. Branching instructions where the condition depends on the tainted data or memory instructions where the data is read from or written into an address calculated using tainted data will be marked for transformation. Transformations are then applied to marked instructions to remove dependencies on tainted data while preserving the original program semantics.

After obtaining the transformed program, we will run it on the BliMe hardware to test the effectiveness of our analysis and transformations. BliMe will report violations at runtime if it encounters branching or memory instructions dependent on tainted data.

3.2 Requirements

Based on the above problem statement, we have identified the following requirements for our analysis and transformations:

3.2.1 Analysis requirements

R-1 Soundness: Analysis should detect all branching and memory access instructions that depend on tainted data.

If the analysis does not report anything, the code should run on BliMe hardware with no violations.

R-2 Precision: Analysis should have a low false-positive rate.

Transformations are applied based on the analysis results, which will cause overhead on the original program. Thus, the number of applied transformations should be minimal.

3.2.2 Transformation requirements

R-3 Correctness-1: Transformations should remove dependencies on tainted data and produce a constant-time program.

R-4 Correctness-2: Transformations should preserve the original program semantics.

Chapter 4

Design

4.1 Overview

Our tool is composed of two parts: analysis and transformations. The analysis phase identifies all branch and memory instructions that depend on secret input. These instructions are then marked for later, where we will apply transformations. Our tools contain three kinds of transformations: branch linearization, load/store linearization, and branchless select.

After our transformations, it is possible that more data structures can become dependent on secret data and can result in new branching or memory addressing instructions depending on secret data. Listing 4.1 describes one such case:

```
1  if (secretData){
2      x = y;
3  }
4  else {
5      x = z;
6  }
7  ....
8  if (x){
9      .....
10 }
```

Listing 4.1: Secret-dependent branch

In the above code sample, the variable `x` does not directly depend on secret data; thus, the second `if` condition is not reported. However, after transformations, the above program

would look like this:

```
1 x = constant_select(secretData, y, z)
2 .....
3 if (x){
4     .....
5 }
```

Listing 4.2: Secret-dependent branch after transformation

`constant_select` is defined through bitwise arithmetic operations that use `secretData` in their operands, which means that the result of the `constant_select` function depends directly on secret data. This causes variable `x` to become dependent on secret data and make the second `if` statement vulnerable, which means that the second if branch should also become linearized.

To handle cases similar to the above, we run our analysis and transformations in a loop. After finishing the transformation, we check whether any transformation modified the IR, and if so, we run the analysis again. Figure 4.1 describes the overall workflow of our tool.

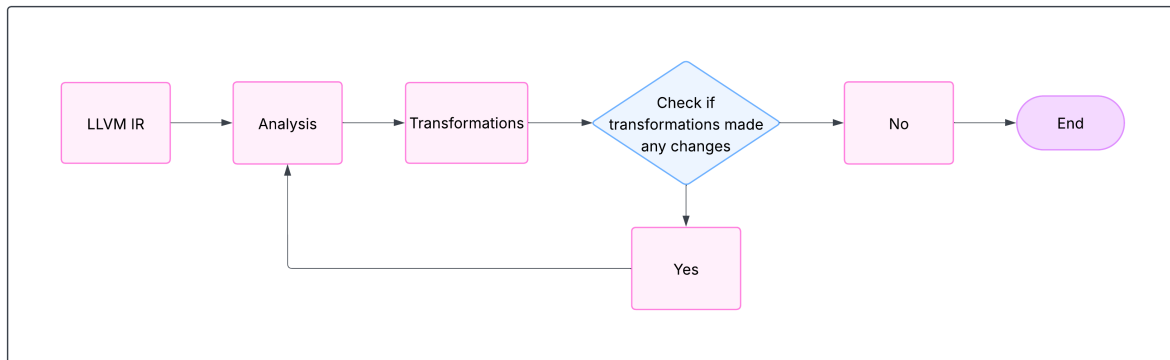


Figure 4.1: Tool workflow

4.2 Analysis

Our goal in the analysis is to track the propagation of sensitive data through the program and determine whether it has been used in branching or memory addressing instructions. We rely on the developer to mark the initial values in the program that contain sensitive data. For the remainder of this section and the ones that follow, we refer to the sensitive data as *tainted* data. Our analysis is based on abstract interpretation, a technique used in static analysis to approximate the program behavior over all potential execution paths. To apply abstract interpretation, we define an abstract domain to represent the values in LLVM and specify abstract semantics to model the concrete semantics of instructions in LLVM IR.

4.2.1 Analysis Workflow

We start the analysis from the `main` function of the program. In LLVM, functions are composed of basic blocks, and we apply our analysis to each of the blocks in the function. At each block, we iterate over the instructions and apply our abstract semantics to them, while keeping a mapping from the LLVM values to abstract states, which represent the total abstract state for the block. Before analyzing each block, we obtain and unify the abstract states of its predecessors to form the initial state mapping for that block.

Blocks are analyzed in topological order, ensuring all predecessors of a block are analyzed before the block itself. Loops form a cyclic control flow, where the loop latch block branches back to the loop header, which serves as the loop’s entry point. In our analysis, we temporarily ignore the backedge from the loop latch to the header to analyze all blocks within the loop body. We continue the analysis until the initial state of the loop entry block does not change between consecutive analysis runs. During the analysis of our loops, we apply a widening operation on the integer values to reach convergence. Algorithm 1 describes how the analysis of loops is performed. For nested loops, we can apply the same logic, where the analysis of inner loops is finished before the analysis of outer loops.

Final analysis results are stored based on each function and the initial abstract state mapping when that function is called, which is referred to as the function signature. The section 4.2.3 describes how this initial state is formed and how these signatures are utilized.

Algorithm 1 Loop State Fixpoint Analysis

```
1: function analyzeLoop(basicBlockArray)
2:   while true do
3:     entryBlock  $\leftarrow$  getLoopEntry(basicBlockArray)
4:     prevEntryState  $\leftarrow$  getInitialStateFromPreds(entryBlock)
5:     for all block  $\in$  basicBlockArray do
6:       analyzeBlock(block)
7:     end for
8:     newEntryState  $\leftarrow$  getInitialStateFromPreds(entryBlock)
9:     if newEntryState = prevEntryState then
10:      break
11:    end if
12:  end while
13: end function
```

4.2.2 Abstract Domain

Abstract Domain is used to approximate the values in LLVM and also contains information on the taintedness of values they represent. In our domain, we can have five different states, which will be described next.

Abstract Integer

The **Abstract Integer** domain represents the integer values in LLVM, regardless of bit width. To approximate the value of an integer, we utilize ranges in our abstract domain, [*lowerBound*, *upperBound*]. This representation means that the original value in LLVM can be any value within the above range. Tracking the value of the integers is important, as these values can be used in indexing operations in arrays, and we want to track what part of the array is being retrieved or updated in **load/store** instructions. Furthermore, as tainted data can only be scalar values in the program, the **Abstract Integer** class also contains information on whether the represented value is tainted or not. This is represented as a Boolean member value in the definition of **Abstract Integer** class.

Abstract Float

The **Abstract Float** domain is used to represent the floating-point numbers in LLVM. However, in our representation, we do not track the actual values of floating-point numbers

due to their complexity, and only track information on whether this value is tainted or not.

Abstract Array

The **Abstract Array** class is used for representing both statically and dynamically allocated arrays in LLVM. Each of the values in the array is treated as a separate abstract state. The **Abstract Array** class contains the following attributes:

- Known elements map: Tracks the abstract states stored at specified array indexes. Since we use ranges to represent integers, array indices are also modeled as ranges.
- Default element: Abstract state that is returned whenever the known elements map does not contain information on the requested element. We use abstract states from the known elements map to construct this state.
- Size: Integer range representing the size of the array for dynamically allocated arrays.

The Listing 4.3 shows what the **Abstract Array** state would look like for the given code snippet.

```
1 int* arr = malloc(sizeof(int) * 10);
2 arr[1] = 5;

// Corresponding LLVM IR:
%size = mul i64 4, 10
%arr = call ptr @malloc(i64 %size)
%elem_ptr = getelementptr inbounds i32, ptr %arr, i64 1
store i32 5, ptr %elem_ptr, align 4

// Abstract Array for %arr:
// Size = (40, 40) (in bytes)
// Default element = Abstract Integer(lowerBound=5, upperBound=5, tainted=False)
// Known elements map = (1, 1) => Abstract Integer(lowerBound=5, upperBound=5, tainted=False)
```

Listing 4.3: Abstract Array example

Abstract Struct

The **Abstract Struct** class represents the struct values in LLVM. It contains a single attribute: a mapping from integer ranges (indices) to the abstract states of struct members.

In LLVM, struct members are accessed using integer indices rather than named fields, as in C/C++.

Abstract Pointer

The `Abstract Pointer` class represents all pointer types in LLVM. The `Abstract Pointer` class contains several attributes, which are described below:

- **Object Content:** Abstract state representing the object that is referenced by the pointer. This state can be any of the following: `Abstract Integer`, `Abstract Float`, `Abstract Array`, and `Abstract Struct`.
- **Pointer Content:** Represents nested pointers. This state is always another `Abstract Pointer` instance.
- **Parents map:** This map represents the other LLVM values that were used to derive the current value that the `Abstract Pointer` class corresponds to. Keys of the map are LLVM values, whereas the values are indexes (integer ranges) that were used to obtain the current pointer
- **GetElementPtr information:** boolean value representing whether this pointer is a result of `getelementptr` instruction or not. This information is needed as the abstract semantics of `getelementptr` and `store` instructions change based on whether we are analyzing regular pointers or pointers derived as a result of `getelementptr` instruction.

The Listing 4.4 describes how the `Abstract Pointer` class is constructed for the given program.

```
1 int* arr = malloc(sizeof(int) * 10);
2 arr[1] = 5;
3 int* data = &arr[1];
```

```
// Corresponding LLVM IR:
%size = mul i64 4, 10
%arr = call ptr @malloc(i64 %size)
%elem_ptr = getelementptr inbounds i32, ptr %arr, i64 1
store i32 5, ptr %elem_ptr, align 4
%data = getelementptr inbounds i32, ptr %arr, i64 1
```

```
// Abstract Pointer for %data:  
// Object Content = Abstract Integer(lowerBound=5, upperBound=5, tainted=False)  
// Pointer Content = NULL  
// Parents map = %arr => (1, 1)  
// GetElementPtr = true
```

Listing 4.4: Abstract Pointer example

Abstract Domain Taint Information

As we discussed above, only primitive values like integers and floats can be tainted (secret) values, which is also represented in our Abstract Domain, where only **Abstract Integers** and **Abstract Floats** have taint information. In addition to this, we also track whether the values in LLVM were derived from a memory addressing instruction where the tainted value was used. This information is part of all the abstract states in our domain, as memory addressing operations can be used to access any kind of data.

In our analysis, it is possible to write tainted data into arrays, structs, or pointers, and later untainted data can be written to those data structures, which will remove the previously tainted data. However, we separate the data structures that were marked tainted by the developer, where writing untainted data to those does not remove the underlying taint. We assume that the data structure marked tainted by the developer will always contain tainted data, and anything derived from those data structures will always be tainted.

4.2.3 Abstract Semantics

This section describes how we model the semantics of various LLVM instructions in our analysis. In LLVM, instructions result in values, and these values are used as keys in our abstract state mapping. We classify the instructions in LLVM into the following categories:

- Arithmetic, Bitwise and Comparison: `add`, `sub`, `fmul`, `and`, `shl`, `icmp`, `fcmp` and others
- Casting: `trunc`, `sext`, `sitofp`, `bitcast`, `inttoptr` and others
- Conditional: `br`, `select`, `phi`
- Memory: `alloca`, `load`, `store`, `getelementptr`
- Call: `call`

Arithmetic, Bitwise and Comparison instructions

For arithmetic and bitwise instructions involving integers, we aim to closely resemble the concrete semantics of the execution. As we represent integers using ranges, we apply the operations on the lower and upper bounds of the operands involved in the operation.

```
// %a = Abstract Integer(lowerBound=-4, upperBound=5, tainted=False)
// %b = Abstract Integer(lowerBound=10, upperBound=12, tainted=True)
%sum = add i32 %a, %b
// %sum = Abstract Integer(lowerBound=-4 + 10, upperBound=5 + 12, tainted=True)
```

Listing 4.5: add instruction example

For all cases where at least one of the operands of the instructions in this category is tainted, we propagate the taint to the resulting abstract state corresponding to the LLVM value. For the comparison instructions, the resulting value is a range represented by `[0, 1]`, which means that the result of the operation can be either true or false.

Casting instructions

For casting between similar types (e.g., `i1` to `i32`), we copy the operand’s abstract state to the result. For casting between different types, we have the following cases:

- Casting between floats and integers: because we do not track values for floats/doubles in our analysis, we only propagate the taint information during the casting operation.
- Casting between pointers and integers: In these cases, we copy the corresponding abstract state to the result state. Here, we can have integer values in LLVM that have **Abstract Pointer** as their state in our domain, which is a mismatch. This is because the value is often cast back to a pointer later, and we aim to preserve the original pointer’s information across such casts. As our analysis does not cover pointer arithmetic, in arithmetic instructions involving such values, we copy the **Abstract Pointer** state to the result of the arithmetic instruction while ignoring its concrete semantics.

Conditional instructions

For **branch** instructions, we only mark the instruction as tainted if the condition is also tainted. For **select** instruction, we unify the states corresponding to the operands and the

end result of the condition is tainted. Similarly, for **phi** instruction, we unify the abstract states of incoming values and assign them to the result of the instruction.

Memory instructions

For memory instructions, our goal is to closely resemble the concrete semantics of the program and track the pointer relations precisely to avoid over-tainting in the analysis. Abstract semantics for these instructions contain the logic for handling complex data structures in the program and detecting memory addressing instructions relying on tainted data.

Load instruction

There can be two cases in the **load** instruction: (1) the loaded value is a scalar value, (2) the loaded value is a pointer value. For the first case, we copy the object content from the **Abstract Pointer** state corresponding to the object content, and in the second case, we copy the pointer content to our resulting state. For the latter case, we create a relation from the resulting state to the operand of the **load** instruction. Since no index is involved, we insert a wildcard entry in the parents map to represent the dependency. Listing 4.6 depicts an example state for this case:

```
%ptr_to_ptr = load ptr, ptr %double_ptr
// Abstract State for %ptr_to_ptr
// .... (other attributes)
// Parents map: %double_ptr => (_, _)
```

Listing 4.6: Load instruction with pointer result

In both cases, we also check whether the address the instruction is loading from has been calculated using tainted values. If so, this instruction is considered a tainted **load** instruction and marked for transformation later.

Store instruction

As with the **load** instruction, the **store** instruction has two cases: storing a scalar or a pointer value. Depending on the type of the value being stored, we update either the object content or the pointer content of the **Abstract Pointer** corresponding to the value in the store instruction. Whenever the program stores a pointer into another pointer, we create a relation between those pointers, similar to the case where a pointer was loaded from another pointer. The same taint propagation rules used for **load** instruction are applied here too. At the end of the abstract semantics of **store** instruction, we iteratively update the parent

values of the destination value to propagate the changes. These updates continue iteratively until a fixpoint is reached (i.e., no new parents are discovered). Listing 4.7 demonstrates the order of the changes applied to abstract states in the given IR example. Here, we are trying to store the value into an integer array that is stored inside a struct.

```
%arr_ptr_in_struct = getelementptr inbounds %struct.S, ptr %mainStruct, i32 0, i32 0
%arr_loaded = load ptr, ptr %arr_ptr_in_struct, align 8
%elem_ptr = getelementptr inbounds i32, ptr %arr_loaded, i64 4
store i32 5, ptr %elem_ptr, align 4

// Order of updates to the abstract states of the LLVM values
// %elem_ptr => %arr_loaded => %arr_ptr_in_struct => %mainStruct
```

Listing 4.7: Store instruction propagating changes

GetElementPtr instruction

In LLVM, the `getelementptr` instruction is used to calculate the addresses where the program wants to load from or store into, which includes accessing elements in an array or member fields of the struct. The base operand of the `getelementptr` instruction is always a pointer, which means we will also have **Abstract Pointer** state in our analysis. We model the `getelementptr` instruction using the following steps:

1. Get the underlying object of the **Abstract Pointer** state
2. Get the relevant data from the object content of **Abstract Pointer** state
 - For arrays - known elements map
 - For structs - fields map
3. Retrieve the abstract state from the map corresponding to the index in the `getelementptr` instruction
4. Wrap the retrieved abstract state in an **Abstract Pointer** state, as the result of the `getelementptr` instruction is always a pointer
5. Mark the result as tainted addressing if the index used was also tainted

Call Instruction

In LLVM, the `call` instruction is used to perform function calls where the target of the function call can be user-defined functions or functions from linked libraries. For the functions that are part of linked libraries where the LLVM IR does not contain the definition, we manually model their behavior in our analysis, such as `malloc` or intrinsic functions in LLVM.

For the user-defined functions, we analyze the called function and return the analysis results to the caller. Before starting the analysis on the called function, we obtain the abstract states corresponding to the function arguments and create a mapping from those values to the abstract states. This mapping serves as a starting point for the analysis of the called function, which is also referred to as the signature of the function call. After the analysis of the called function has finished, the results of the analysis are stored in the global map, which the caller function uses to retrieve the results from. Before analyzing the called function, we check if this signature has been analyzed before, and if so, we directly return the results without performing the analysis again.

If the called function has non-void return type, we obtain the `ret` instruction in the called function and get the abstract state corresponding to the returned value. This abstract state is assigned to the LLVM value that performed the function call. Furthermore, if any of the passed arguments to the called function is a pointer, we get the final abstract state for the relevant argument from the function analysis results. Then, the abstract state for the pointer that was passed in the function call is updated with the abstract state retrieved from the called function. The Listing 4.8 describes this case:

```
define dso_local void @callee(ptr noundef %p) {
entry:
    store i32 5, %p
    ret void
}

define dso_local i32 @caller() {
entry:
    %x = alloca i32, align 4
    store i32 7, ptr %x, align 4
    ; %x = Abstract Pointer(object content = Abstract Integer(lb=7, up=7, tainted=False))
    call void @callee(ptr noundef %x)
    ; %x = Abstract Pointer(object content = Abstract Integer(lb=5, up=5, tainted=False))
    ret i32 0
}
```

Listing 4.8: Function with pointer arguments

4.3 Transformations

In our transformations, we target branching and memory addressing instructions. In LLVM IR, these correspond to the `branch`, `select`, `load`, and `store` instructions. After obtaining the analysis results, transformations are applied and the analysis is rerun. This process continues until no transformations can be applied.

4.3.1 Select Instruction

LLVM `select` instruction performs selection between two values based on a condition. If the condition of the `select` instruction is tainted, it may result in secret-dependent branching in the generated assembly. Through the use of bitwise arithmetic, `select` instructions can be transformed into a series of bitwise operations. For a given `select` instruction with `cond` as condition, `trueVal` as the value to be selected in the true case, and `falseVal` as the value to be selected in the false case, we apply the following steps to select the correct value without the use of the `select` instruction:

1. Obtain the negation of the condition
2. Extend the size of the negated condition to match the size of the selected values
If the condition is true, this will result in all 1s, otherwise it will be all 0s.
3. Compute the bitwise `xor` of the `trueVal` and the `falseVal`
4. Compute the bitwise `and` between the result of step 2 and 3
5. Compute the bitwise `xor` of the result from step 4 and `falseVal`

If the condition of the select is true, the result of the step 4 will be $(\text{trueVal} \oplus \text{falseVal})$. XORing this result with `falseVal` yields `trueVal`, which is the desired value. In case of condition being false, step 4 will result in all 0s, and the last step will just produce the `falseVal`.

4.3.2 Branch Instruction

In LLVM IR, `branch` instructions can be either conditional or unconditional, where the conditional branches can lead to one of the two possible paths. In our transformations, whenever the condition of the `branch` instruction becomes tainted, we aim to execute both

of the branches, i.e., turn conditional branches into unconditional branches and select the right values in the end. Aside from executing both sides of the branch, more transformations might be needed if the tainted branch is part of a loop structure, as it can possibly control the number of times the loop has been executed.

In a conditional **branch** instruction, a program can take two different paths, and then those paths will merge at another basic block later in the program, namely **mergeBlock**. During branch linearization, we sequentially execute the first path originating from the branch instruction and then directly connect it to the second path, making sure that all the instructions in both paths are executed regardless of the condition. Thus, we assume that the paths from the conditional branch to the **mergeBlock** do not intersect, meaning they do not share any basic blocks, to prevent any cycles from being formed during linearization. However, this assumption does not always hold, as demonstrated in Listing 4.9:

```
1  if (a || b) {  
2      doSomething();  
3  } else {  
4      doNothing();  
5  }  
  
entry:  
    br i1 %a, label %if.true, label %check.b  
  
check.b:  
    br i1 %b, label %if.true, label %if.false  
  
if.true:  
    call void @doSomething()  
    br label %if.end  
  
if.false:  
    call void @doNothing()  
    br label %if.end  
  
if.end:  
    ret void
```

Listing 4.9: Intersection paths

In the above example, the **branch** instruction in **entry** block can lead to the following paths, where **if.end** block is the merging point:

- True path: **if.true** → **if.end**

- False path 1: `check.b` $\rightarrow$ `if.true` $\rightarrow$ `if.end`
- False path 2: `check.b` $\rightarrow$ `if.false` $\rightarrow$ `if.end`

Here, `if.true` block is shared between the True Path and False Path 1. If we try to linearize the `branch` instruction in `entry` block, the True Path would connect to the False Path 1, meaning `if.true` block will branch into `check.b` block. However, because the `check.b` block itself branches into `if.true` block, this linearization can create an infinite loop in the program.

To avoid this issue, we pre-process the LLVM IR before passing it into the analysis and transformation. During this pre-processing, for every conditional `branch` instruction, we check whether the paths from the `branch` instruction to the merge point have any shared blocks. If so, we collect all the shared blocks, create copies of those blocks, and replace the original blocks in one of the paths with the copied blocks, as shown in Listing 4.10.

```
entry:
  br i1 %a, label %if.true, label %check.b

check.b:
  br i1 %b, label %if.true.copy, label %if.false

if.true:
  call void @doSomething()
  br label %if.end

if.true.copy:
  call void @doSomething()
  br label %if.end

if.false:
  call void @doNothing()
  br label %if.end

if.end:
  ret void
```

Listing 4.10: Intersection paths removed

After linearizing the above IR, we will have two calls to the function `doSomething`, which does not match the semantics of the original program. However, the execution of the functions and their effects will be guarded by the original condition variable used to determine where to branch, which will be described in the following section.

Non-Loop-Control Branch Transformations

Given the following **branch** instruction `-br i1 %cond, label %trueBlock, label %falseBlock-` we want to linearize control flow so that both blocks are executed, and correct results are propagated to the rest of the program. In a simple case where assuming the blocks will merge at a common successor block `%mergeBlock`, we will have the following control flow path:

```
br label %trueBlock (original branch instruction)
br label %falseBlock (end of the %trueBlock)
br label %mergeBlock (end of the %falseBlock)
```

After transforming the control flow of the program, as both blocks will be executed, we first inline the function calls happening inside both blocks to be able to guard their effects in the program. Then, we update the **store** instructions in both blocks, including the ones coming from inlined functions, where the values in the true block (false block) will only be stored if the condition is true (false), otherwise we load the existing value from the destination and store it back to the destination again.

The above transformations describe a simple **branch** instruction where true and false blocks merge immediately after at the end of their instructions, which is not always the case. For example, in Listing 4.11, both true and false blocks lead to two other blocks before merging:

```
1  if (taintedCond){
2      if (globalValue > 5){
3          x = 1;
4      }
5      else {
6          x = 2;
7      }
8  }
9  else {
10     if (globalValue > 5){
11         x = 3;
12     }
13     else {
14         x = 4;
15     }
16 }
17 return x;
```

Listing 4.11: Blocks merging later

To handle cases like this and have a more generalized approach, we apply the following steps in our transformation for the given `branch` instruction:

```
br i1 %cond, label %trueBlock, label %falseBlock
```

1. Find the merge point of `%trueBlock` and `%falseBlock`, referred as `%mergeBlock`
2. Find all paths from `%trueBlock` to the `%mergeBlock`, referred as `truePaths`
3. Find all paths from `%falseBlock` to the `%mergeBlock`, referred as `falsePaths`
4. For every path in `truePaths`, update the last block to branch to `%falseBlock` instead of `%mergeBlock`
5. Transform every store instruction in blocks in `truePaths` and `falsePaths`

Loop-Control Branch Transformations

There are two cases where a branch can affect the execution count of the loop: (1) loop control condition is tainted branch, (2) tainted branch inside the loop leads to a block outside the loop (`break` statement can cause this). In both cases, the loop control branch is replaced with a custom branch that executes the loop a given number of times, and that number will be set to a constant provided by the developer. This is done to make sure that tainted branch, whether it comes from the loop header (where the loop control branch is located) or inside the loop, does not affect the execution count of the loop.

Case 1: tainted loop control

As mentioned above, the first step is to replace the loop control branch with a custom branch that executes the loop a given number of times. However, we will also need to take into account that the loop body should only be executed if the loop control condition evaluates to true to preserve the original program semantics. Listing 4.12 describes how this transformation would look like in a simple example:

```
1 // Before
2 for (int i = 0; i < taintedVal; i++){
3     ....
4 }
5
6 // After
7 int i = 0;
```

```

8  for (int count = 0; count < constantIterationCount; count++){
9      if (i < taintedVal){ // Tainted branch
10         ....
11         i++;
12     }
13 }

```

Listing 4.12: Tainted value in loop condition

However, after the above transformation, we still have a tainted branch inside the loop. Now, we can treat this branch as a non-loop-control branch, and apply the same transformations that were described earlier.

Case 2: tainted branch exiting loop

Similar to the previous case, loop control branch is replaced with a branch that executes the loop given number of times, and the original loop control condition is used to determine whether the instructions inside the loop should be executed. However, in this case, we will also need to keep track of the branch condition that exits the loop. If that condition has been satisfied, we have to make sure that the instructions inside the loop are not going to be executed, even if the original loop control condition is satisfied, to preserve the original program semantics. Listing 4.13 describes how this transformation would look like:

```

1  // Before
2  for (int i = 0; i < unTaintedVal; i++){
3      if (taintedVal){break;}
4      .... // code after break
5  }
6
7  // After
8  bool breakFlag = false;
9  for (int count = 0, i = 0; count < constantIterationCount; count++){
10     if (i < unTaintedVal && breakFlag == false){ // Tainted branch
11         if (taintedVal){ // Tainted branch
12             breakFlag = true;
13         }
14         if (breakFlag == false){
15             .... // code after break
16         }
17         i++;
18     }
19 }

```

Listing 4.13: Tainted branch breaking loop

After the transformations, we can see that there are still two branches that are tainted. Starting from the inner tainted branch, we apply the transformations from non-loop-control branch transformations to linearize the control flow and propagate the right values for the rest of the program.

Similar to the non-loop-control branches, there can be more complicated cases where the tainted branch inside the loop does not directly exit the loop, but the blocks it leads to eventually exit the loop, shown in Listing 4.14:

```
1  for (int i = 0; i < untaintedVal; i++){  
2      if (taintedCond){  
3          if (globalValue > 5){  
4              x = 1;  
5              break;  
6          }  
7          else {  
8              x = 2;  
9          }  
10     }  
11 }
```

Listing 4.14: Tainted branch can exit the loop later

Therefore, to have a more generalized approach, we apply the following steps for tainted branches that can finish the loop early:

```
br i1 %taintedCond, label %loopNonLeavingBlock, label %loopLeavingBlock
```

1. Find the merge point of the blocks in the branch.
2. If the merge point is outside the loop, this means one of the blocks in the branch is not part of loop.
 - (a) Transform the branch into unconditional branch to %loopNonLeavingBlock
 - (b) Save the branch condition for modifying the loop control later
3. If the merge point is part of the loop, this means both blocks are inside the loop.
4. Find all branches that can leave the loop on the path from %loopLeavingBlock to merge point
5. For all the branches found in step 4, starting from the last one, apply steps 1 and 2
6. Modify the loop control to iterate for a constant number of times

7. Use the conditions collected from step 2(b) and original loop condition to update `store` instructions inside the loop

Algorithm 1 describes the full flow of linearizing tainted branches in a given function. The following are the descriptions of some of the variables and functions used in the algorithm:

- `loopsToBranches`: mapping from loop to boolean values, where the boolean values come from branches inside the loop that can cause early exit
- `handleNonLoopControlBranch`: refers to process described in section non-loop-control branch transformations [4.3.2](#)
- `getAnd`: perform and operation between given conditions
- `createBranch`: we create a new basic block where the original loop control block branches into. Based on the result of the provided condition, we either go into the loop body, or back to the loop control

Algorithm 2 Branch Linearization

```
1: function linearizeBranch(branchInstruction)
2:   if isNonLoopControlBranch(branchInstruction) then
3:     handleNonLoopControlBranch(branchInstruction)
4:   else
5:      $loop \leftarrow \text{getCorrespondingLoop}(\text{branchInstruction})$ 
6:     if isLoopControlBranch(branchInstruction) then
7:        $loopsToBranches[loop] \leftarrow loopsToBranches[loop] \cup \{branchInstruction\}$ 
8:     else
9:        $blockA \leftarrow \text{getSuccessor}(\text{branchInstruction}, 0)$ 
10:       $blockB \leftarrow \text{getSuccessor}(\text{branchInstruction}, 1)$ 
11:      if blockA and blockB do not leave the loop then
12:        handleNonLoopControlBranch(branchInstruction)
13:      else
14:         $merge \leftarrow \text{getMergePoint}(blockA, blockB)$ 
15:         $leavingBranchesA \leftarrow \text{getLeavingBranches}(blockA, merge)$ 
16:         $leavingBranchesB \leftarrow \text{getLeavingBranches}(blockB, merge)$ 
17:        fixLeavingBranches(leavingBranchesA)
18:        fixLeavingBranches(leavingBranchesB)
19:        handleNonLoopControlBranch(branchInstruction)
20:      end if
21:    end if
22:  end if
23: end function

24: function linearizeAllBranches(branches)
25:    $loopsToBranches \leftarrow \{\}$ 
26:   for all branch  $\in$  branches in reverse DFS order do
27:     linearizeBranch(branch)
28:   end for
29:   for all loop  $\in$  keys( $loopsToBranches$ ) do
30:      $conditions \leftarrow loopsToBranches[loop]$ 
31:     updateLoopControlToConstant(loop)
32:      $conditions \leftarrow conditions \cup \{\text{getOriginalCondition}(loop)\}$ 
33:      $andCondition \leftarrow \text{getAnd}(conditions)$ 
34:      $newBranch \leftarrow \text{createBranch}(andCondition, loopBody, loopControlBlock)$ 
35:     handleNonLoopControlBranch(newBranch)
36:   end for
37: end function
```

4.3.3 Load and Store Instructions

In LLVM IR, `load` and `store` instructions are used to read and write data from statically and dynamically allocated objects. If the tainted values are used for addresses used in these instructions, the program can become vulnerable to cache-based timing attacks and leak sensitive information. To prevent these from happening, we transform the `load` instructions by loading the data from every possible address derived from the base pointer used in original `load` instruction, and only propagate the result of the load that corresponds to the correct index using `select` instruction. We applied the same idea to the `store` instructions, where we load from every possible address, and use `select` instruction to either store the value from the original `store` instruction or the newly loaded value based on the index being accessed. Listing 4.15 demonstrates what the code would look like after the transformation:

```
1 // Before
2 int value = arr[taintedIdx];
3
4 // After
5 int value = -1;
6 for (int i = 0; i < arrSize; i++){
7     int cur = arr[i];
8     value = select(i == taintedIdx, cur, value);
9 }
```

Listing 4.15: Transforming load instruction

Listing 4.16 describes the similar transformation for the `store` instruction:

```
1 // Before
2 arr[taintedIdx] = value;
3
4 // After
5 for (int i = 0; i < arrSize; i++){
6     int cur = arr[i];
7     arr[i] = select(i == taintedIdx, value, cur);
8 }
```

Listing 4.16: Transforming store instruction

For arrays that span multiple dimensions, the above logic can still be applied with modifications. We would still iterate over all the elements in the dimension where the tainted index was used, while keeping the values in other dimensions constant. Listing 4.17 shows how this transformation would look in two dimensions:

```

1 // Before
2 int value = arr[taintedIdx][nonTaintedIdx]
3
4 // After
5 int value = -1;
6 for (int i = 0; i < arrSize; i++){
7     int cur = arr[i][nonTaintedIdx];
8     value = select(i == taintedIdx, cur, value);
9 }

```

Listing 4.17: Transforming load instruction in multiple dimensions

To properly apply our `load` and `store` transformation, we need to know the size of the array from which the original `load` and `store` derived the address. For statically allocated objects, this information is available in the LLVM IR, can be obtained using the relevant API functions. In case of dynamically allocated objects, we rely on the information gathered during the analysis. Whenever the `malloc` function is used to allocate memory, we store the value of the argument passed to the `malloc` call. Then, when the `getelementptr` instruction is used to calculate the address using the allocated object as base pointer, we calculate the size based on the stored value passed to the `malloc` call and the type of the value retrieved by the `getelementptr` instruction.

Chapter 5

Implementation

5.1 Analysis

The analysis is implemented in LLVM 17.0 using the LLVM Pass Infrastructure and is registered as a Module Pass. We build our target program using WLLVM (Whole-program LLVM). By using WLLVM, we make sure that all the relevant files to the program are combined into a single IR file that can be analyzed by our pass. During the build, we turn off all the optimizations to make sure all the information is preserved and instructions are used in a simpler manner (for example, breaking up a single `getelementptr` instruction to multiple instructions, which makes it easier to track pointer relations). After obtaining the LLVM IR file, we run several passes on the IR to reduce redundant memory operations, normalize loops so that they have single entry and single exit, and unify the return instructions in functions.

5.1.1 Introducing taint

For our analysis, we ask the developer to mark the data that contains secret information, i.e., tainted data. We introduce a new attribute called `blinded`, which the developer can use to mark global variables or function arguments as tainted. Furthermore, local variables inside the class can also be tainted through the use of dummy helper functions. The Listing 5.1 demonstrates how this can be done for custom structs initialized inside functions as local variables.

```

1 void initializeStruct(customStruct* myStruct);
2 void __attribute__((noinline)) taintHelper(__attribute__((blinded))
   customStruct* myStruct){}
3
4 int main(){
5     customStruct myStruct;
6     initializeStruct(&myStruct);
7     taintHelper(&myStruct);
8 }

```

```

define dso_local noinline void @taintHelper(ptr noundef blinded %myStruct) {
entry:
    ret void
}

define dso_local i32 @main() {
entry:
    %myStruct = alloca %struct.customStruct, align 4
    call void @initializeStruct(ptr noundef %myStruct)
    call void @taintHelper(ptr noundef %myStruct)
    ret i32 0
}

```

Listing 5.1: Tainting local variables

From the above LLVM IR example, when the analysis is being performed on `taintHelper` function, it will taint the argument of the function due to the `blinded` attribute. After the analysis is done on `taintHelper` function and it returns to the main function, we propagate the taint information to the local variable `%myStruct` due to it being a pointer.

5.1.2 Abstract Domains

We implement our Abstract Domains through polymorphic classes in C++. All types of domains are based on an abstract state class which contains information on whether the current abstract state is from a tainted memory addressing instruction, and whether it has been tainted by `blinded` attribute. The abstract domains are then divided into two groups: Abstract Objects, which include `Abstract Integer`, `Abstract Float`, `Abstract Array`, `Abstract Struct`, and `Abstract Pointer`. These domains are used to approximate

the LLVM values, which include instructions inside functions and global variables. Below we briefly describe each domain:

- **Abstract Integer**
 - value: IntegerRange, (int, int)
 - tainted: boolean
- **Abstract Float**
 - tainted: boolean
- **Abstract Array**
 - size: IntegerRange, (int, int)
 - defaultElement: AbstractState*
 - knownElements: map<IntegerRange, AbstractState*>
- **Abstract Struct**
 - fields: map<IntegerRange, AbstractState*>
- **Abstract Pointer**
 - objectContent: AbstractObject*
 - pointerContent: AbstractPointer*
 - isGepPointer: boolean
 - parents: map<LLVM::Value*, IntegerRange>

5.1.3 Analysis Workflow

Our analysis starts by going over global variables in the module and constructing appropriate states for them. Then, we iterate over the functions available in the module to find the `main` function which will be the start point of our analysis. Analysis starts from the `main` function and continues until it reaches the `ret` instruction. Every time a new function needs to be analyzed, `analyzeFunction` is invoked, which in turn invokes `analyzeSCC` for loops and `analyzeBlock` for single basic blocks. Our analysis is performed on the basic blocks of the function in topologically sorted order, which is obtained by the `scc_iterator` utility

provided by the LLVM. Each basic block has an associated state mapping that describes the abstract states of the LLVM values in that and preceding blocks, which is defined as `map<LLVM::Value*, AbstractState*>`. At the end of each basic block, this mapping is stored, which will be used for constructing the initial state for the successors of the current block.

To construct the initial state for analyzing a basic block, we unify the abstract state mappings collected from its predecessor basic blocks. In the unification process, for the LLVM values (`LLVM::Value*`) that only exist in a single basic block, they are just added to the combined state mapping as they are. For the values that appear in multiple basic blocks (for example, an `if` and `else` statement where we store different values into an integer pointer), we unify their corresponding abstract states. This unification process happens recursively, where for complex abstract states like `AbstractArray`, `AbstractStruct`, and `AbstractPointer`, we apply unification to the abstract states contained inside those states. For integers, the unification operation merges the intervals, picking the lowest lower bound as the new lower bound and the highest upper bound as the new upper bound. For both integers and floats, if any of the states included in the unification are tainted, the result of the unification will also become tainted. The same logic applies to the propagation of tainted addressing information.

For each basic block, we iterate over its instructions, determine their types using `Instruction.getOpcode()`, and apply the appropriate abstract semantics. In some of the instructions, such as casting, we copy the abstract state from the operand of the instruction to the result of the instruction. This copying is done through creating a new abstract state pointer and recursively populating it from the information obtained from the abstract state corresponding to the operand. This is done to prevent aliasing between `AbstractState*`, which can cause issues in the analysis.

For `call` instructions, analysis first constructs the initial state using the passed arguments and the current abstract state mapping. Then the analysis invokes the `analyzeFunction` recursively with the constructed initial state. When the analysis reaches the `ret` instruction, we store the state mapping at that point as the result of the analysis. These results are then retrieved by using the corresponding `LLVM::Function` object and initial state mapping.

5.2 Transformation

We implemented our transformations as part of the LLVM Pass infrastructure. Our transformation pass invokes the analysis pass and obtains the analysis results. Based on the

results, we apply branch transformations first, since they may introduce new tainted load instructions. Then the analysis is run again, and load and store transformations are applied to the updated results. Lastly, we apply transformations to `select` instructions. This process is repeated in a loop until no transformations can be made.

5.2.1 Select Transformation

As discussed in Section 4.3.1, we use bitwise arithmetic to replace `select` instructions. In our description, we talked about `select` instructions, where the selected values were assumed to be integer. However, it is also possible that `select` instruction is used to select between floating point or pointer values.

Case 1: floating point values

LLVM provides `bitcast` instruction, which can be used to interpret the value as floating point value without changing the underlying bitwise representation. When selecting between floats, we use `bitcast` on the values in the `select` instruction to get integer values, apply our bitwise arithmetic operations, and in the end convert the result back to the original type using `bitcast` instruction.

Case 2: pointer values

Similar to the above, LLVM provides `ptrtoint` and `inttoptr` instructions that can be used to convert values to and from pointers. Like the `bitcast` instruction, `ptrtoint` instruction interprets the address in the pointer as an integer value without changing the bitwise representation. After applying our transformations, we convert the value back to a pointer using the `ptrtoint` instruction.

5.2.2 Branch Transformations

After receiving the analysis results, we extract the branches where the condition has been tainted. As these transformations are function-level transformations, we group the tainted branches based on the functions they are in. Then, we apply Depth-First Traversal to order the branches and start applying our transformations in reverse.

During the transformations, we linearize the control flow by executing both blocks controlled by the `branch` instruction, and picking the right values based on the branch condition. However, if there are any `phi` instructions present in the merging of these

blocks, an incorrect result will be picked, as the `phi` instruction decides on which value to pick based on which block the execution was coming from. To prevent this from happening, before the transformations are applied, we traverse the instructions in each function, and use `DemotePHIToStack` utility provided by LLVM to remove any `phi` instructions.

Non-Loop-Control Branch Transformations

For branches that do not control loops, we first find the merge point of the blocks in the `branch` instruction using graph traversal algorithms. Then we collect the blocks on the path from each basic block in the `branch` instruction to the merge point. We iterate over instructions in these collected basic blocks, and use the LLVM `InlineFunction` utility to inline the function calls. After function inlining, we iterate over the instructions again, and transform the `store` instructions based on the original condition of the `branch` instruction.

Loop-Control Branch Transformations

To determine whether our current `branch` instruction and the basic block it resides in is a part of a loop structure, we rely on the `LoopAnalysis` pass provided by the LLVM. Using this pass, we can obtain the innermost loop information (represented by `Loop` class) for a given basic block. This loop information can be used to obtain the header of the loop (`Loop.getHeader()`), which is the basic block that is part of the loop and dominates all the other blocks in the loop. The loop information can also be used to check whether any given block is part of the loop (`Loop.contains()`). Because we modify the control flow of the program in our branch transformations, this can change information regarding the loop. To make sure that the loop information is up to date, after every transformation affecting program control flow, we run the `LoopAnalysis` again to receive updated loop information.

For each of the basic blocks in the tainted `branch` instruction, we traverse the blocks reachable from them to check whether they can exit the loop they are part of without going through the loop header. This means that the `branch` instruction can potentially cause early exit from the loop, affecting loop iteration count. If both basic blocks do not cause an early exit, we treat this case as non-loop-control branch transformations.

For the `branch` instructions that can cause early exit, we perform another graph traversal from the successors of the branch to the loop header. During this traversal, we collect all the `branch` instructions where one of the successors is not part of the loop. Then, going in the reverse order, for each `branch` we modify the conditional branch to an unconditional

branch and treat it as a non-loop-control branch transformation where the merge point is the loop header. We also create a boolean value that stores the condition of the original `branch` to know whether the break has happened or not.

After removing and modifying all the branches that can cause early exit, we transform the loop header to run for a constant number of times, where the number is provided by the developer. Finally, the original loop control condition and all the booleans created for this loop are combined in `and` instruction, where the result of this instruction is used to guard all the `store` instructions inside the loop.

5.2.3 Load and Store Instructions

For `load` and `store` instructions, we remove the original instructions and use the `splitBasicBlock` function to split the block at the `load/store` instruction to `beforeBlock` and `afterBlock`. Then, we create an additional basic block (called `loopBodyBlock`) where we iterate over all the possible values of the base array and select the correct value based on the original index. We then link the `beforeBlock` to `loopBodyBlock` and `loopBodyBlock` to `afterBlock`. LLVM provides `replaceAllUsesWith` function, which we use to replace the original `load` instruction with the selected value in our `loopBodyBlock` block.

Chapter 6

Evaluation

In this chapter, we evaluate the effectiveness of our analysis and transformation passes. Initially, we will run the selected programs through our analysis and BliMe hardware simulation, and compare the results. For evaluation, we use a version of BliMe that does not crash the target program, but only reports the violations which include secret dependent branching and memory addressing instructions. Then, we will run the programs through our transformation pass and obtain modified IR. This modified IR will be run on BliMe again to check whether the transformation pass have successfully mitigated side-channel vulnerable code.

For evaluation, we use programs from the OISA [31] benchmark files. In the OISA paper, the authors provide ISA extensions that allow tracking of sensitive data and also oblivious instructions to handle sensitive data in a safe manner. They provide a benchmark of well-known data processing algorithms to test their approach, which we will also be using to test our analysis and transformations.

6.1 Analysis results

Table 6.1 summarizes the results of the analysis and BliMe simulation.

Table 6.1: Number of distinct violations reported

Test	Only Analysis		Only BliMe		Both	
	Memory Access	Branch	Memory Access	Branch	Memory Access	Branch
binary_search	0	0	0	0	0	2
dijkstra	2	0	0	0	8	6
dnn	0	0	2	3	0	2
find_max	0	0	0	0	0	1
int_sort	0	0	0	1	0	2
kmeans	0	0	0	0	0	2
matrix_mult	0	0	0	0	0	0
page_rank	0	0	0	0	5	0
PQ	0	0	0	1	0	4

6.1.1 R-1 Soundness

Table 6.1 shows that for most of the programs, the number of violations detected by the analysis is either equal or more than the violations reported from BliMe. After individually analysing each violation and the code location it corresponds to, we determined that for 6 of the programs all the violations reported by the BliMe simulation were also detected by the analysis.

For the `int_sort`, `PQ` and `dnn` benchmark, one of the branch violations came from the `_int_free` function. This function is called as a result of the `free` function call, which is part of the pre-compiled library, `glibc`. Thus, the generated LLVM IR only has the declaration of the `free` function, and does not have the definition to be analyzed. Similarly, the rest of the violations on the `dnn` benchmark come from `exp` function from the pre-compiled `math` library, which again was not part of the generated LLVM IR that was provided to the analysis. To prevent these violations from happening, we can substitute `math` library’s `exp` function with our custom, constant-time version.

Based on the above results, we can say that the analysis satisfies the soundness requirement, detecting all the violations reported by BliMe, provided that the input IR file contains the definition of the function where the violations happen.

6.1.2 R-2 Precision

Table 6.1 shows that in only one benchmark file, analysis reports more violations than the hardware. Upon inspecting this case, we found that the following code snippet corresponds to the additional memory violations reported by the analysis:

```
1 int v = graph->adjlist[u][i];    // data dependent
2 int weight = graph->weights[u][i]; // data dependent
```

In the above code snippet, the variable `u` is tainted, and BliMe reports one load violation for each line. Below is the corresponding LLVM IR for the first line in above code:

```
%adjlist = getelementptr inbounds %struct.Graph, ptr %16, i32 0, i32 1, !dbg !741
%17 = load ptr, ptr %adjlist, align 8, !dbg !741
%18 = load i32, ptr %u, align 4, !dbg !742
%idxprom12 = sext i32 %18 to i64, !dbg !740
%arrayidx13 = getelementptr inbounds ptr, ptr %17, i64 %idxprom12, !dbg !740
%19 = load ptr, ptr %arrayidx13, align 8, !dbg !740 ; reported by analysis
%20 = load i32, ptr %i6, align 4, !dbg !743
%idxprom14 = sext i32 %20 to i64, !dbg !740
%arrayidx15 = getelementptr inbounds i32, ptr %19, i64 %idxprom14, !dbg !740
%21 = load i32, ptr %arrayidx15, align 4, !dbg !740 ; reported by analysis
```

In the above IR, the analysis first reports violation for the `load` instruction corresponding to the value `%19`, as the tainted value is used to derive the address used in `load`. However, the analysis also reports violation for the `load` instruction corresponding to the value `%21`. This happens, because the base pointer used in `getelementptr` instruction in the line before is `%19`, pointer loaded from tainted address, and because analysis considers the value `%19` itself to be a 'tainted address', any address calculated from it will result in another tainted address, causing another violation. Similar behavior happens for the second line in the above C code, resulting in two additional memory violations in total.

Based on the above results, our analysis reported extra violations in only one benchmark, meaning it has low false positive rate, satisfying the precision requirement.

6.2 Transformation results

After applying the transformations on the target programs, we run them again in the BliMe simulation and record the reported violations. Table 6.2 summarizes the results after transformations:

Table 6.2: Number of distinct violations reported by Analysis and BliMe after transformations

Test	Results from Analysis		Results from BliMe	
	Memory Access	Branch	Memory Access	Branch
binary_search	0	0	0	0
dijkstra	0	0	0	0
dnn	0	0	2	4
find_max	0	0	0	0
int_sort	0	0	0	1
kmeans	0	0	0	1
matrix_mult	0	0	0	0
page_rank	0	0	0	0
PQ	0	0	0	1

6.2.1 R-3 Correctness-1: Removing non-constant time code

After running our transformed programs on the BliMe simulation, 5 of the programs did not report any violations. PQ, kmeans, int_sort each report 1 branching violations, which comes from the `_int_free` function, as it cannot be transformed. For dnn, one of the violations comes from the `_int_free` function, while the rest originate from the `exp` function. Neither of these functions is part of the input LLVM IR; thus, they cannot be transformed to remove the violations.

Based on the above results, we can say that the transformations can successfully remove timing-based side-channel vulnerable code, provided that the input IR file contains the definition of the function where the initial vulnerable code regions reside.

6.2.2 R-4 Correctness-2: Preserving program semantics

For our tool to be usable, programs should preserve their original semantics after the transformations. To test this property, we run the programs with the same inputs before and after transformations and compare the outputs. For the benchmarks that had loop transformations, we manually bounded the loops with the maximum number of iterations the loop can have. From our observations, transformed programs preserved the original semantics and produced the expected outputs.

6.3 Performance evaluation

We also measured the processing time for each of the benchmark files and the overhead introduced by our transformations. Table 6.3 describes the time it took for our analysis and transformation loop to finish. Most of the benchmarks finished under a second, and `dijkstra`, `int_sort`, PQ benchmarks took more than 10 seconds due analysis and transformations happening in multiple rounds and having relatively larger code bases.

Program	Processing time (sec)
<code>binary_search</code>	0.17
<code>dijkstra</code>	26.2
<code>dnn</code>	0.12
<code>find_max</code>	0.02
<code>kmeans</code>	1.88
<code>int_sort</code>	25.25
<code>matrix_mult</code>	0.02
<code>page_rank</code>	0.15
PQ	12.57

Table 6.3: Time taken by analysis and transformation passes

Program	Original	Transformed
<code>binary_search</code>	0.003	0.003
<code>dijkstra</code>	0.003	0.648
<code>dnn</code>	0.007	0.007
<code>find_max</code>	0.002	0.002
<code>kmeans</code>	0.020	0.048
<code>int_sort</code>	0.003	0.424
<code>page_rank</code>	0.003	0.014
PQ	0.002	0.003

Table 6.4: Execution time of programs in seconds

Table 6.4 shows the runtime of the programs before and after transformations are applied. `dijkstra` and `int_sort` shows significant overhead, 220 and 140 times respectively. For the `dijkstra` benchmark, manual transformations from OISA paper [31], add around 800 times overhead to the execution. When evaluated against larger data size (4096 nodes

for `dijkstra` algorithm) our transformation added 800 times overhead compared to 6800 times overhead from OISA.

For the `int_sort` benchmark, manual transformations from OISA paper achieves better performance (10 times overhead), as their manual transformations utilize the parallel processing and understanding of program semantics. Another contributor to the overhead comes from the condition of the loop in merging algorithm, shown in Listing 6.1. During transformation, this loop is changed to run for the provided upper bound times, which is the complete input size, contributing to the overhead.

```
1 while (i < len_L && j < len_R) // tainted condition
```

Listing 6.1: `int_sort` main loop control

Chapter 7

Discussion

In this work, we developed a static analyzer to detect timing-based side-channel vulnerable code, and implemented program transformations that modify the code to remove these vulnerabilities. We tested the effectiveness of our approach by running our analysis and transformations on a set of programs that implement data processing algorithms, and validated our results against the BliMe architecture.

We have implemented our tool as an LLVM Pass, allowing for easy integration into LLVM-based build pipelines. Developers can add annotations in their programs to mark the secret data and invoke our transformation pass, which will run the analysis and apply program transformations, producing IR that is timing-based side-channel safe.

Additionally, our tool can be used in conjunction with the BliMe architecture. As BliMe will abort any program execution where the private data flows into branching or memory addressing instructions, our tool can be used to generate a BliMe-safe binary, making it easier to utilize the protections offered by BliMe.

7.1 Limitations

7.1.1 Load and Store Transformations

In our transformations, we remove the vulnerabilities arising from memory addressing instructions, `load` and `store`, by iterating over all the possible indexes in the target array, and selecting the correct `load` result or `store` destination using `select` instruction where the original index is used in the condition (this `select` instruction is transformed later

too). However, for this transformation to be successful, we need to know the size of the array that is used in memory addressing instructions. For programs in our evaluation, we were able to obtain this information from either LLVM IR for statically allocated arrays, or from the analysis where we tracked the integer value passed to `malloc` or `calloc` functions. However, in more complex programs, where the arguments of memory allocation functions are undetermined, possibly due to dependence on user input, we would not be able to apply our transformations. One potential solution for this is to have the developer to include the array size information in the program and pass this information to the analysis, either through using special naming conventions to indicate which variables describe sizes for which arrays or by using annotations to convey this information.

7.1.2 Loop execution counts

For branch transformations, where the tainted branch could affect the loop execution count, we modified the loop condition to make the loop execute a set number of times. This number should be provided by the developer to make sure that the loop will run the appropriate number of times to preserve the original program semantics. As there can be many such loops, this process can require high manual effort, decreasing the usability of our tool. However, the developers of cryptography libraries are usually aware of issues brought forth by secret data being used in loop conditions, and during implementation, they avoid using variable-time loops and try to rely on constant loop bounds.

7.1.3 Pre-compiled libraries

As described in the evaluation section, even after the transformations, we still had violations reported from the BliMe simulation due to standard library or math library calls. This happened because the input LLVM IR did not contain the definitions of the functions where the violations were coming from, so they could not be transformed. A possible solution for this is to have the developer implement and use the constant-time versions of those library functions to avoid any potential leakage.

Chapter 8

Related works

Side-channel attacks aim to leak sensitive information by utilizing additional information produced by the system during the execution of the program, such as execution time or power consumption, and have been a long-standing problem in security and privacy. In this work, we primarily focus on timing attacks that arise from certain coding patterns, such as secret-dependent branching or secret-dependent memory addressing. Initially theorized in 1996 by Paul C. Kocher [14] and demonstrated practically in 2003 by David Brumley and Dan Boneh [8], there has been a body of work on detecting and mitigating timing side-channel attacks in software.

Side-channel vulnerable code detection. Several program analysis techniques have been employed to detect side-channel vulnerable code. CaSym [6] utilizes symbolic execution to track both program states and abstract cache states, and explores all possible paths in the program. It detects side-channel vulnerabilities by checking if changing the secret inputs leads to different cache states, which can leak sensitive information. However, CaSym focuses on cache-side-channel attacks and does not detect timing differences that can result from branches executing different instructions based on secret data. Similarly, SpecuSym [10] and SpecSafe [7] use symbolic execution and static analysis for discovering side-channel vulnerabilities; however, they target side-channel attacks resulting from speculative execution.

DiffFuzz [19] uses dynamic analysis techniques to find side-channel vulnerabilities. It executes the program under different secret inputs and measure the resource usage (time, memory) differences and prioritizes inputs that result in a higher difference. CacheQL [32] also utilizes dynamic analysis, collecting execution traces from the programs using Intel Pin and analysing the relationship between the secret inputs and the collected traces. Then

it uses techniques from cooperative game theory to identify instructions that contribute most to the sensitive information leakage.

Side-channel Mitigation/Program Transformations. SC-Eliminator [29] uses static analysis to detect secret-dependent branches and memory accesses and applies program transformations to mitigate them. It employs taint tracking to monitor propagation of secret values through the program, and then uses static cache analysis to reduce the number of memory access instructions to be mitigated. However, they perform intraprocedural analysis, which requires each function to be manually annotated for secret inputs. LIF [25] also uses static analysis to detect secret-dependent branches and determine symbolic bounds for arrays. They primarily focus on secret-dependent branches and transform only the memory access instructions affected by the secret-dependent branch.

Constantine [5] uses both static and dynamic analysis techniques to detect secret-dependent code and later apply control flow linearization (CFL) and data flow linearization (DFL) to mitigate side-channel vulnerabilities. It also uses information from dynamic profiling for loops whose execution count depends on secret data.

DSA [17] builds upon the model checker CBMC [15], and introduces the concept of differential sets, where for each memory access depending on secret data or a secret-dependent branch, it collects all the possible addresses that could be accessed. Then, to mitigate vulnerable memory access, the program is transformed to access every address in the differential set. Similar to CaSym [6], DSA also focuses on cache side-channel attacks and does not consider timing attacks.

Pendulum [22] relies on fuzzing to identify a public input and a set of secret inputs that can leak sensitive data. Later, it collects the basic blocks that are affected by the secret inputs and applies pattern-based code repair techniques. Pendulum repeats this process until the vulnerability is mitigated or the quantified value of vulnerability is reduced enough. Pendulum mainly focuses on detecting and mitigating timing side-channel attacks and does not address vulnerabilities arising from secret-dependent memory access instructions.

References

- [1] Cve-2024-1544. <https://www.cve.org/CVERecord?id=CVE-2024-1544>.
- [2] Cve-2024-23170. <https://www.cve.org/CVERecord?id=CVE-2024-23170>.
- [3] Cve-2024-2467. <https://www.cve.org/CVERecord?id=CVE-2024-2467>.
- [4] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in kyber implementations. Cryptology ePrint Archive, Paper 2024/1049, 2024.
- [5] Pietro Borrello, Daniele Cono D’Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS ’21*. ACM, November 2021.
- [6] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 505–521, 2019.
- [7] Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, and Gang Tan. Spec-safe: detecting cache side channels in a speculative world. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.
- [8] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12, SSYM’03*, page 1, USA, 2003. USENIX Association.

- [9] Hossam ElAtali, Lachlan J. Gunn, Hans Liljestrand, and N. Asokan. BliMe: Verifiably secure outsourced computation with hardware-enforced taint tracking. In *Proceedings 2024 Network and Distributed System Security Symposium*, NDSS 2024. Internet Society, 2024.
- [10] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. Specusym: speculative symbolic execution for cache timing leak detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1235–1247, New York, NY, USA, 2020. Association for Computing Machinery.
- [11] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. “they’re not that hard to mitigate”: What cryptographic library developers think about timing attacks. Cryptology ePrint Archive, Paper 2021/1650, 2021.
- [12] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. Flop: Breaking the apple m3 cpu via false load output predictions. In *USENIX Security*, 2025.
- [13] Jason Kim, Daniel Genkin, and Yuval Yarom. Slap: Data speculation attacks via load address prediction on apple silicon. In *S&P*, 2025.
- [14] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
- [15] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, April 2014.
- [16] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.
- [17] Cong Ma, Dinghao Wu, Gang Tan, Mahmut Taylan Kandemir, and Danfeng Zhang. Quantifying and mitigating cache side channel leakage with differential set. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

- [18] Marina Minkin and Baris Kasikci. Zipchannel: Cache side-channel vulnerabilities in compression algorithms. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 223–237, 2024.
- [19] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. Diffuzz: Differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 176–187, 2019.
- [20] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. Cryptology ePrint Archive, Paper 2005/271, 2005.
- [21] Moinuddin K. Qureshi. Ceaser: mitigating conflict-based cache attacks via encrypted-address and remapping. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 775–787. IEEE Press, 2018.
- [22] Haifeng Ruan, Yannic Noller, Saeid Tizpaz-Niari, Sudipta Chattopadhyay, and Abhik Roychoudhury. Timing side-channel mitigation via automated program repair. *ACM Trans. Softw. Eng. Methodol.*, 33(8), November 2024.
- [23] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical timing side-channel attacks on memory compression. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1186–1203, 2023.
- [24] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, Santa Clara, CA, August 2019. USENIX Association.
- [25] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. Side-channel elimination via partial control-flow linearization. *ACM Trans. Program. Lang. Syst.*, 45(2), June 2023.
- [26] Luigi Soares and Fernando Magno Quintão Pereira. Memory-safe elimination of side channels. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 200–210, 2021.
- [27] Han Wang, Hossein Sayadi, Tinoosh Mohsenin, Liang Zhao, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Mitigating cache-based side-channel attacks through randomization: A comprehensive system and architecture level analysis. In

2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1414–1419, 2020.

- [28] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 675–692, Santa Clara, CA, August 2019. USENIX Association.
- [29] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 15–26, New York, NY, USA, 2018. Association for Computing Machinery.
- [30] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. Cryptology ePrint Archive, Paper 2013/448, 2013.
- [31] Jiyong Yu, Lucas Hsiung, Mohamad El’Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *The Network and Distributed System Security Symposium (NDSS)*, 2019.
- [32] Yuanyuan Yuan, Zhibo Liu, and Shuai Wang. CacheQL: Quantifying and localizing cache Side-Channel vulnerabilities in production software. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2009–2026, Anaheim, CA, August 2023. USENIX Association.