Prevention of timing-based micro-architectural attacks

A Project Report

submitted by

NITHESH N. HARIHARAN

in partial fulfilment of the requirements
for the award of the degree of

DUAL DEGREE (B.Tech. and M.Tech.)

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2021

THESIS CERTIFICATE

This is to certify that the thesis titled **Prevention of timing-based micro-architectural attacks**, submitted by **Nithesh N. Hariharan**, to the Indian Institute of Technology, Madras, for the award of the degree of **DUAL DEGREE (B.Tech. and M.Tech.)**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Chester Rebeiro
Research Guide
Professor
Dept. of Computer Science Engineer-
ing
IIT Madras, 600 036

Place: Chennai

Date: 12 June 2021

ACKNOWLEDGEMENTS

I would like to thank Prof. Chester Rebeiro for guiding me through the project, and Keerthi K. for helping me while working on the project. I would like to thank all my professors at IITM for helping me through my journey. I would also like to thank my parents and all my friends for the constant support.

ABSTRACT

We explore popular micro-architectural attacks, and tools commonly used to detect such attacks. We explore advantages and disadvantages of existing work on this area, and propose a new solution which would aid with preventing timing attacks on programs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Micro-architectural attacks are becoming very popular and a focus of a lot of security researches off late. Timing based attacks typically have the attacker use the running time of the program on a variety of inputs to deduce some secret values in the program. The timing differences could occur because of multiple reasons. Two common things that cause timing differences are cache accesses (hit and a miss result in very different runtimes) and instruction count differences (caused by branches, loops, etc). Based on knowledge of the algorithm and the implementation, attackers try to extract secret information by just using runtime.

The 2 popular attacks in this domain are Meltdown and Spectre. Meltdown has some really good solutions to prevent it, while some of the variants of Spectre have been prevented. There are more variants of Spectre that get discovered every so often, which bypasses all the previous solutions that were built up.

The other attacks use techniques like the "prime and probe" to exploit the cache timing, or just plain timing based if you know the internal code and architecture (e.g. if you know a certain condition forces the algorithm to early quit, that can be used).

Cloud and distributed computing make these attacks really powerful because now if you have different virtual machines on a cloud server, they might share the same hardware. If one of the virtual machines is compromised, the attacker could possibly use the fact that both share the same hardware to time the actions performed by the other VM and decipher any secrets.

Typically, the secrets that attackers are interested in are encryption keys, which (depending on the algorithm) tend to be 128 or 256-bit keys. If the attacker would like to bruteforce the key, it would take 2^{256} tries. But, if the attacker can get any kind of information or signal from the runtime, it tends to drastically reduce the search space. We shall see this in an example later.

We will explore existing solutions which aim to prevent such attacks, and the pros and cons of each of these solutions. We will also look at possible performance improvements for these solutions, and impacts on security by trying to improve performance. Finally, I propose a new solution which would provide security against timing attacks. Also, will mention my work on trying to prevent meltdown style attacks using a static analysis pass in LLVM.

1.1 Description of the attacks, with examples

1.1.1 Instruction based timing attacks

These type of attacks rely just on the programming running, and taking different times based on inputs. There are a variety of scenarios that cause the program runtime to vary. These include if clauses without else, or unbalanced if and else block lengths, loops which depend on the input to terminate, or calls to functions whose runtime is dependent on the inputs. These attacks depend a lot on the knowledge of the underlying algorithm.

These attacks could be prevented by making all code paths take time independent of the secret variable, which was written as a transformation pass in LLVM as explained by Wu *et al.* (2018). However, this causes a huge performance hit. We will explore this existing solution in a further chapter.

Some common examples of this type of attack are mentioned below. Also mentioning some equivalent timing attack free code. These examples are inspired from (Wikipedia contributors, 2021).

```
int strcmpVulnerable(string s1, string s2) {
    if (strlen(s1) != strlen(s2))
        return 0;
    for (int i = 0; i < strlen(s1); i++)
        if(s1[i] != s2[i])
            return 0;
    return 1;
}
```

We see that the if the 2 strings have different sizes only the first if clause is executed.

This would mean the runtime is very short in comparison to an execution where the sizes are the same. This fact can be exploited to figure out information about either s1 or s2. Implementations like this could have wide ranging consequences, like if this function is used in a password checker, it would hypothetically be possible to brute force the password using just the runtime.

Another version of this, which is still vulnerable would be as follows.

```
int strcmpVulnerable2(string s1, string s2) {
    int areEqual = 1;
    if (strlen(s1) != strlen(s2))
        areEqual = 0;
    for (int i = 0; i < strlen(s1); i++)
        if(s1[i] != s2[i])
            areEqual = 0;
    return areEqual;
}
```

Here, even if the strings have different lengths, the return is only at the end. Which might deceive you to think that this program is not vulnerable to timing attacks. Notice that if the strings are different lengths, the statement inside the if block gets executed, while if they are the same length, then this statement is not executed. Same argument for the if in the for loop. This again results in a difference in runtimes based on the code path taken.

In this simple example itself we see how cumbersome it gets to fix this problem. In a large codebase, these timing differences could arise from interactions between multiple functions, and it would become impossible ensure the full codebase is free of timing attacks. The focus of most research in this area is to ensure all encryption algorithms which are standards, are timing attack free. This includes multiple implementations of the same algorithm.

1.1.2 Cache timing attacks

In the previous section we saw how the number of instructions run affects time taken and can be exploited. Another major problem that gets exploited in side channel attacks is the fact that there is a huge difference in time taken by a cache hit and a cache miss. Typically, cache accesses happens several orders of magnitudes faster than a main memory access, which will result in a stark difference in the runtime.

Typically in these attacks we assume the attacker has access to perform basic cache operations like flush, or loading the cache with some values. This can be done easily if the cache architecture is known, as the attacker can access particular memory addresses and it will do the required operation. There are many popular attacks in this domain, and we'll briefly look at them. Some examples include 'prime and probe' and 'flush and reload'.

Flush and Reload attack

In this attack, the attacker should be able to flush the cache. Assume a particular instruction or data entry needed by the program is cached. If the attacker flushes the cache, it gets removed, and now when the program accesses that memory address again, it is going to take significantly longer for the program to run. Careful use of the flush operation gives the attacker valuable information about secrets.

Prime and probe attacks

This is a different approach where the attacker sets up the cache with some data (called the prime phase) and then probes the cache. During the probe phase, some of the data previously loaded might be replaced, which will increase the access time.

All the cache based attacks rely a lot on the algorithm. What information you would like to get depends completely on the algorithm. A strategy to retrieve keys from AES encryption using cache timing attacks was explained in the first paper for cache timing attacks. The attacker exploits inherent knowledge about OpenSSL's AES encryption implementation, and uses that to guess the key byte by byte. This shows how these attacks allow attackers to reduce the space to search for the key using the auxiliary information (runtime) .

The other 2 very popular cache attacks include meltdown and spectre Kocher *et al.* (2019). These 2 demand separate attention, so we will dedicate a section for them.

1.1.3 Meltdown

Lipp *et al.* (2018) first describes an attack called Meltdown which allows attackers to breach the memory barriers between user space and kernel space and access any memory location. This is possible due to the behaviour of the cache alongside out-of-order execution which is implemented on processors.

The typical idea in meltdown is to access a memory location that is not accessible, and use the data to access a part of a large array. That part of the array would be cached during out of order execution, and later the program would actually error out. But then if you re-access the array that you tried to access before, only one block would have low access time, as it got cached already. This can be used to figure out what the data byte in the memory location was, and hence repeating this for different memory addresses, you can get a full dump of the memory.

Now any program that was running in other threads would have stored variables and other data in the RAM. Even kernel uses the same RAM. If the attacker gets a dump of the RAM, basically any secret variables used in any program can be extracted from the RAM.

To patch this vulnerability, Intel had added memory fences, which forces ordering of instructions around statements where these instructions are used. By preventing out of order execution, such vulnerabilities are prevented.

However, this poses a severe performance impact as the pipeline would be empty during these instructions. In a later section, we will discuss a possible strategy using static LLVM passes to detect such memory leaks.

CHAPTER 2

LLVM

2.1 LLVM

A lot of further discussions of both existing and new solutions rely on LLVM introduced by (Lattner and Adve, 2004). LLVM provides a set of tools which allow compiler operations to be performed on code. You can use it to build a front-end for an existing language, or create a back end for a supported instruction set. It also comes with a full set of tools like clang, opt, a debugger, etc.

LLVM operations are performed on a language independent representation called the LLVM IR (LLVM intermediate representation). This is a strongly typed language that can be generated using clang. This is then processed using LLVM passes written in C++, which can be either analysis or transformation. An analysis pass just looks at the IR to determine some characteristics. A transformation pass can convert the existing IR to a new IR which can then be converted to an executable to be run. We will see both these kinds of passes which help us tackle the timing attack issues.

2.1.1 LLVM IR

We will generate the IR from a c code. Assuming you have llvm, clang and opt installed, we can then run the following command on a program to obtain the LLVM IR. Lets take a simple program in c that just creates 3 variables and prints it.

```c
#include <stdio.h>

int main() {
    int a = 5;
    int b = 10;
    int c = a + b;
    printf("%d", c);
    return 0;
}
```

Now we run the following command to get the LLVM IR for the code.

```
clang -S -O0 -emit-llvm hello.c -o hello.ll
```

We get the following LLVM IR output. We will understand the various aspects of the LLVM IR.

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %2, align 4
  store i32 10, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %5, %6
  store i32 %7, i32* %4, align 4
  %8 = load i32, i32* %4, align 4
  %9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x
    i8], [3 x i8]* @.str, i32 0, i32 0), i32 %8)
  ret i32 0
}
```

We see that it creates 4 variables, of type i32 which is a 32 bit integer. These are referenced using %1, %2 and so on. It then has the store instructions to load 3 of the variables. Then it perfoms the add operation by using temporary variables %5 and %6. It then stores the result to the correct variable. We then see it call the print statement, and finally return 0.

The IR also supports a variety of branching instructions and comparison instructions.

2.1.2 LLVM passes

We then move on to understand how LLVM passes work, and the general structure of an LLVM pass. LLVM provides us with a pass manager class, which works in conjunction with the opt tool to run the pass. Note, the following discussions use the old LLVM pass manager which was replaced by a new pass manager in the new LLVM versions. However, as all online documentation still reference the old pass manager, and there is no significant resources for the new pass manager, we had implemented passes using the old pass manager.

We code up the passes using C++, and the LLVM libraries. There are a lot of useful data structures, and functions that can make processing LLVM IR very easy.

To begin with, you need to create a subclass of a Pass class provided by LLVM. This could be one of FunctionPass, ModulePass, LoopPass, RegionPass, and some more provided by the LLVM library. Each of these trigger differently, for e.g. the FunctionPass runs the pass per function. So you'll need to pick the largest sub-unit of the program that you would like to start navigating.

The subclass needs to satisfy a few properties. Firstly, it needs an ID field, which allows opt to identify the pass uniquely. It needs to have a constructor, which calls the baseClass' constructor with the ID. It needs a runOn* method. For example, for a FunctionPass, you'll have a runOnFunction with the following signature.

```
bool runOnFunction(Function &F);
```

Similarly, you'll have a runOnModule or runOnLoop and so on for each type of pass. The return value denotes whether the pass modifies the Control Flow Graph (CFG) or not. This is important, because the passManager needs to know this.

Finally, outside the class, you need to use RegisterPass with the PassName to register the pass to be used by opt. This c++ file can then be compiled to a .so file, which can then be used directly in opt to run the pass on a particular .ll file which can be generated as per the instructions above.

2.1.3 An Analysis Pass Example

At the start of the project, while learning LLVM, I had worked on an assignment problem, and this section can be used by anyone trying to learn LLVM to quickly grasp all the basics.

The problem statement was to analyse a given program with for loops to ensure all variables in the initialization block of the for loop are used in the condition block of the for loop. All input for loops will be preceeded with a call to printf immediately before. Here is the code that performs this operation, and after that we'll break it down to understand how it works.

```cpp
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"
#include "llvm/Support/raw_ostream.h"
#include <llvm/IR/Instructions.h>
#include <llvm/Analysis/LoopInfo.h>
#include <llvm/IR/InstrTypes.h>
#include <map>
#include <vector>
#include <set>

using namespace llvm;
using namespace std;

namespace {
struct LoopMultipleInitPass : public FunctionPass {
  static char ID;
  LoopMultipleInitPass() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override {
    LoopInfo &LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
    int loopCount = 0;
    for (LoopInfo::iterator i = LI.begin(), e = LI.end(); i != e; i
  ++) {
      Loop *l = *i;
      set<Value *> values;

      set<Value *> valuesUsed;
      vector<string> valuesUnused;
      set<Value *> multipleInitValues;

      // Get the basic block just before the loop.
      BasicBlock *preheader = l->getLoopPreheader();
      int call_found = 0;
      for (BasicBlock::iterator bb = preheader->begin(); bb !=
  preheader->end(); bb++) {

    // In the multiple init, we care about all store instructions
    if (call_found && strcmp(bb->getOpcodeName(), "store") == 0) {
      // The 2nd operand of the store instruction has the variable data.
      Value* val = cast<Value>(bb->getOperand(1));
      multipleInitValues.insert(val);
    }

    // Wait till we find the print call (which identifies the start of the multiple init section).
    if (strcmp(bb->getOpcodeName(), "call") == 0) {
      call_found = 1;
    }
  }

  // Get the loop header (condition block)
  BasicBlock *header = l->getHeader();
  for (BasicBlock::iterator bb = header->begin(); bb != header->end(); bb++) {
    Value *val = cast<Value>(bb);

    // Add the result to values used.
    valuesUsed.insert(val);

    // Add all operands to values used.
    for (int k = 0; k < bb->getNumOperands(); k++) {
      Value *v = bb->getOperand(k);
      valuesUsed.insert(v);
    }
  }

  // Check that all elements in the multiple init set is present in the values used set.
  for (auto it = multipleInitValues.begin(); it != multipleInitValues.end(); it++) {
    if (valuesUsed.find(*it) == valuesUsed.end()) {
      valuesUnused.push_back((*it)->getName());
    }
```

```
69       }
70       if (valuesUnused.size() > 0) {
71         errs() << "No"<< "\n";
72         for (auto elem: valuesUnused) {
73           errs() << elem << " ";
74         }
75       } else {
76         errs() << "Yes" << "\n";
77         for (auto elem: multipleInitValues) {
78           errs() << elem->getName() << " ";
79         }
80       }
81       errs() << "\n";
82     }
83     return false;
84   }
85 
86   void getAnalysisUsage(AnalysisUsage &AU) const override {
87     AU.setPreservesCFG();
88     AU.addRequired<LoopInfoWrapperPass>();
89   }
90 };
91 }
92 
93 char LoopMultipleInitPass::ID = 0;
94 
95 static RegisterPass<LoopMultipleInitPass>
96     X("loop-multiple-init-pass", "Loop Multiple init pass",
97       true, false);
```

A loopInfo object needes to be created using the getLoopInfo() function. This object provides useful information about loops in the program. This Object is an iteratable collection. As we see on line 21, we first generate a LoopInfo object. Then we loop over the loop info object. At each loop, we first get the Loop object (Line 24).

The whole program is divided into basic blocks. Each basic block may contain 1 or more instructions. BasicBlocks are very commonly used in passes. Similar to Assembly, each instruction has an Opcode. The Opcodes uniquely define the operation being performed.

The loop object has some useful utilities. We use the getLoopPreheader function in line 32 to get the preheader block for the loop. This is the block just before the loop header. The header is the block that defines the condition (this is where control comes back to in the branch statement at the end of the loop).

The preheader contains all the initialization statements, and the call statement, and might contain statements above the call (this is as per the input structure defined). By looping over the preheader basic block (line 34), it iterates over each instruction. The iterator bb is analogous to an instruction object. Instruction objects have useful methods like getOpcodeName and getOperand(OperandPosition) which we use. In line 44, we detect a call instruction, which marks the printf call. If we have found the printf call, then we care about store instructions. These instructions will be of the form "store variableName value". The target variable, which gets initialized will be the 1st operand. GetOperand gets a pointer (which is of type Instruction). We Cast it to a Value type (which is what LLVM uses to represent operands).

We make a dictionary of all the operands which are used in the initialization section. Next we get the LoopHeader, which has the condition blocks. We then insert all operands used in the conditions. Operands can be on both LHS and RHS of a condition. We add all operands to the dictionary.

Finally, we Loop through the dictionaries to match. If any element is not used, we print it. We see that we can call getName() to get the variable name as it is in the code.

CHAPTER 3

Analysis of existing solutions

There is a lot of existing literature that outlines solutions that help detect and prevent timing based attacks. Each of them have some advantages, and disadvantages. Some of these use LLVM based passes to process the data. We will look at these solutions in detail in the below sections.

3.1 Eliminating Timing Side-Channel Leaks using Program Repair

This section is based on work by Wu *et al.* (2018). This is a pure LLVM based pass, which transforms the program to eliminate timing attacks. The algorithm here guarantees that the input program is time invariant to changes in secret variables. It aims to eliminate both instruction and cache based timing attacks relating to secret variables.

The Authors also use a variety of inputs to prove correctness (the number of cycles taken by the program is equal for different secret inputs). It uses a variety of interesting techniques to detect sensitive branches and make sure that the sensitive code is time invariant. We'll explore those techniques below.

3.1.1 Unbalanced conditional branches

Suppose you have an if-else block which switches based on the secret, and the lengths of the if and else block are different, then this is a potential leak. If for some configuration of the secret, the shorter block runs, then this is valuable information gained by the attacker. The goal should involve making the branches depending on the secret the same size (size meaning number of instructions).

Here is an example of a vulnerable program and the mitigation used.

```
if(a[0]&1) b[2] |= 1; // leak
if(a[1]&1) b[1] |= 1; // leak
if(a[2]&1) b[0] |= 1; // leak
```

The first mitigation suggested was to balance it by using an else block, where you assign a dummy variable. however, this solution when tested for time independence failed for different secrets. This might possibly be due to caching effects differing the number of cycles needed for accessing the dummy variable and the actual variable. Due to all these microarchitectural effects coming into play, the author then proceeds to define an LLVM intrinsic called CTSEL, which performs the same operation as above, but does it in a time independent fashion. The true implementation depends on the architecture, but the LLVM version would be as follows.

```
b[2] = CTSEL(a[0]&1, b[2]|1, b[2]);
b[1] = CTSEL(a[1]&1, b[1]|1, b[1]);
b[0] = CTSEL(a[2]&1, b[0]|1, b[0]);
```

This has no branches, and depending on the first value, will either assign the second or the third value to the Left hand side. This is the final strategy used to remove branch statements, which can potentially cause timing leaks.

3.1.2 Inconsistent cache accesses

This is caused typically because most cryptography algorithms implementations use a lookup table for the sbox. Now if the full lookup table is present in the cache, all accesses will take the same time, and there will be no timing leaks. If we can ensure that all elements are in the cache before we look it up, it will work. The author outlines 3 countermeasures possible, each with some advantages and disadvantages.

We would like to prevent a timing leak for a line of the form

```
block[i] = sbox[block[i]];
```

As a first mitigation, we access all elements in the sbox everytime. If it is not in the cache, it will incur a miss and load that block into the cache.

```
block_i = block[i];
for (j=0; j < 256; j++) {
    sbox_j = sbox[j];
    val = (block_i == j)? sbox_j : block_i;
}
block[i] = val;
```

This does 255 calls which are pointless, and has a new branching instruction which depends on $block_i$ which typically would depend on the secret value. This will prevent the timing leak but at a really large performance hit.

The second mitigation strategy, assuming you know the size of a cache block in the architecture the program is running on, is to access 1 element from each block to ensure the block gets cached. This is more efficient compared to the above, and achieves the same objective.

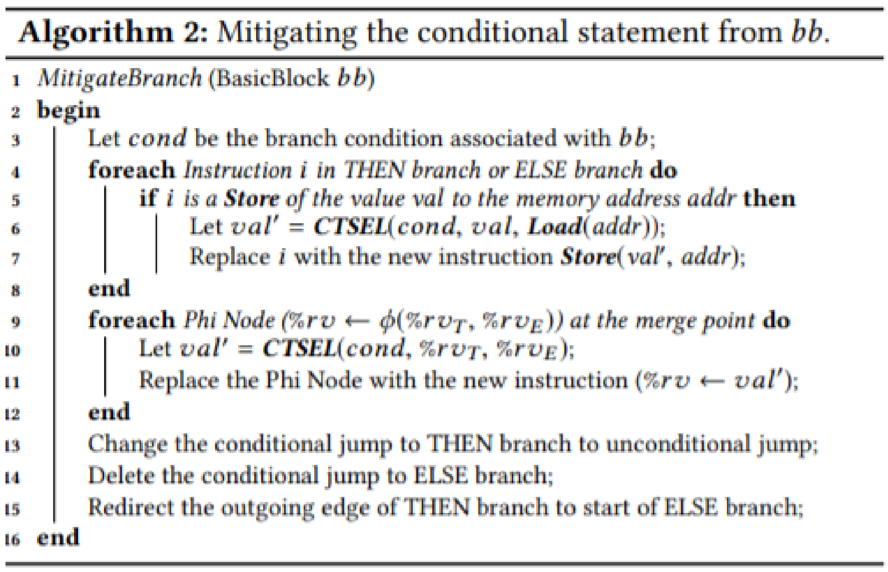
```
block_i = block[i];
for (j=block_i % CLS; j < 256; j+=CLS) {
    sbox_j = sbox[j];
    val = (block_i == j)? sbox_j : block_i;
}
block[i] = val;
```

The last mitigation strategy is to use a pre-loading loop, where all blocks get cached and then we access normally. If the program for sure doesn't remove these elements from the cache after the pre-loading loop, this will work, assuming no external interference. If, for example, the attacker has access to flush the cache, and decides to flush just after the pre-loading loop is done, the program becomes vulnerable again. So this is a really efficient but risky approach.

```
for (j =0; j < 256; j+=CLS)
    temp = sbox[j];

//access to sbox[...] is always a hit
block[i] = sbox[block[i]];
```

3.1.3 Detecting timing leaks using an LLVM pass

The paper now suggests a strategy to detect possible timing leaks using an LLVM pass on the IR. The pass expects a list of secret variables as the inputs. Any variable directly computed using the secret variable, or uses conditions on the secret variables will be



Algorithm 2: Mitigating the conditional statement from bb.

```
MitigateBranch (BasicBlock $bb$)
begin
    Let $cond$ be the branch condition associated with $bb$;
    foreach Instruction $i$ in THEN branch or ELSE branch do
        if $i$ is a Store of the value val to the memory address addr then
            Let $val' = \textbf{CTSEL}(cond, val, \textbf{Load}(addr))$;
            Replace $i$ with the new instruction $\textbf{Store}(val', addr)$;
    end
    foreach Phi Node $(\%rv \leftarrow \phi(\%rv_T, \%rv_E))$ at the merge point do
        Let $val' = \textbf{CTSEL}(cond, \%rv_T, \%rv_E)$;
        Replace the Phi Node with the new instruction $(\%rv \leftarrow val')$;
    end
    Change the conditional jump to THEN branch to unconditional jump;
    Delete the conditional jump to ELSE branch;
    Redirect the outgoing edge of THEN branch to start of ELSE branch;
end
```

Figure 3.1: LLVM pass to mitigate sensitive conditional branches.

marked as sensitive. They also approach pointer structures differently by marking specific fields as sensitive instead of the full array. As an example, if a[0] is dependent on the secret, only a[0] is sensitive and not a.

Now, once all sensitive variables are determined, a sensitive branch is where the condition depends on the sensitive variables. Such branches get replaced by CTSEL as mentioned above. The other sensitive instructions are accesses to lookup tables, where the index is a sensitive variable. This algorithm is demonstrated in Figure 3.1 (taken from the original paper by Meng Wu).

Similar approach is taken for the sensitive lookup table accesses. As the author assumes a weak attacker model, where the attacker only gets to observe the runtime of the program, he goes ahead with the risky, but efficient pre-loading strategy mentioned in the previous section. He also proceeds to discuss a strategy to decide using static analysis to determine if a particular instruction will surely hit cache, in which case the pre-loading is not necessary. This would not be possible if you assume a stronger attack scenario.

3.1.4 Advantages, disadvantages, and possible improvements

This algorithm definitely does make the timing leaks non-existent in programs. However, the weak attacker assumption results in only a small section of attacks being countered by this algorithm. Even if the attacker just executes another program in parallel which can interfere with the cache, timing leaks can be caused.

However, as the algorithm works to ensure equal CPU cycles regardless of the secret, most statistics based attacks are mitigated (where attackers time multiple runs to deduce secrets).

We also notice in the results that even with such relaxed assumptions on the attackers, this algorithm suffers from a 3-10x slowdown (across different algorithms). Also, the transformed code is 5-10x larger than the normal executable. This could be concerning for small budget devices, which would like to use cryptography algorithms as they have both limited space and compute power.

This approach is as optimized as it can be, for the assumed threat model. As the transformation operations are only performed on sensitive branches or lookup table accesses, if we try to gamble and not optimize any of these sensitive operations, it can cause a timing leak by carefully designing inputs.

Another issue is this approach has been tailored to fit cryptography algorithms. If the program uses any external functions (library functions maybe), which depend on the sensitive variables, then the LLVM pass will fail. The programmer needs to be careful to avoid using any such functions. It also depends on the size of the input look up table. If a new program has a variable sized table, then this runs into problems. It does solve the timing leak issue in crypto libraries, but on a larger scale cannot protect programs from timing leaks. There might be non-crypto programs which would like to protect their variables from timing attacks, but they can't really benefit from this approach due to the lack of generalization.

In conclusion, this work provides a lot of insights into detecting timing leaks through static analysis and prevention of weak attacks by making the program runtime independent of the sensitive data. It works really well in practice when used on cryptography algorithms like AES. The code can still be vulnerable to attacks like Meltdown, Spectre, and attacks where the attacker can control the cache (either through flush, load, etc). The performance hit and executable size is significantly increased, but attempting to reduce either of these results in a potential leak.

3.2 ct-fuzz: Fuzzing for Timing Leaks

This paper by He *et al.* (2020) dives into a very different approach to prevent timing leaks. They use fuzzers which are typically used to detech memory leaks by testing complex software. They use these fuzzers to detect issues where 2 different runs of the same program might express a property (say runtime) differently. Thus this would be able to detect a program vulnerable to timing attacks.

The core idea that ct-fuzz uses is to execute 2 copies of a given program, in isolation so both don't interfere, and then the program crashes if the 2 executions are distinguishable based on the test property. For example, if the 2 executions take significantly different time, then the program would crash.

This approach uses a fuzzer which uses some randomization and a fixed algorithm to generate new inputs from existing inputs. The ct-fuzz algorithm manages to detect timing vulnerabilities, but does not aim to solve the timing vulnerability problem.

It is a very efficient solution to detect, as it uses intelligently crafted inputs to detect a timing leak. However, its utility seems limited as it would be impossible as a programmer to manually fix all possible timing leaks so as to "pass" the test posed by ct-fuzz. This is the key disadvantage of this method. A key advantage is that with suitable tools, this methodology can be used to detect any leaks, where the expected behaviour is constant across multiple runs. For example, if you can measure the power consumed by the 2 programs, this can be used to detect attacks that use the power side channel.

3.3 An Efficient Mitigation Method for Timing Side-Channels on the Web

This paper by Schinzel (2011) goes into a few possible strategies which can be used to prevent timing side-channel attacks. They look at it from a web perspective, where the client would measure time taken between a request and a response. Though not strictly cache related, these ideas are interesting to look at.

First, he proposes fixing a time T, which is greater than all possible runtimes of the program, and send a response at T. This way the attacker will only observe T. He then proceeds to discard this approach as for most programs, worst case runtime would be much worse than the average runtime. The performance hit would be too high.

Next, he proposes adding a random delay at the end of the program before sending the packet. This though sounds reasonable, is not particularly secure. Consider the same random number generator is used to get the random number you decide to wait for. This will mean that the random delay you add gets sampled from the same distribution each time. If possible, the attacker could sample from this distribution infinitely and figure out reasonable estimates for the noise. Once you know the noise distribution, it will be possible to estimate the true values using a de-noising logic (something like MAP estimation). Thus, using simple statistics, it would be possible to deduce the true timing characteristics assuming the attacker is allowed to collect enough samples.

Finally, he proposes a really nice idea to prevent timing attacks, which is to use a fixed hash function which uses the inputs and the secrets and outputs the amount of time to wait for. This is really nice because now the wait time is a function of the input and the secret. Its not a random number that can be estimated statistically. This is going to give a fixed delay to the execution, and its going to be really hard to reverse engineer either the true runtime or the secret inputs. The only way the attacker can figure stuff out is to figure out the hash function used. A good hash function is not reversible, i.e. given the output, there is no way to know what the input (or a range of possible inputs) is. In this situation, the attacker doesn't know the hash output. He only knows the sum of the hash output and runtime of the program, and there is no way to separate these 2 values.

This solution would however be vulnerable to a power side channel attack if the attacker had the means to access the power traces. When the thread sleeps, the power consumption would be a flat curve. So that wouldn't really help as you can time the execution from the start of the power spike to the end. But assuming that the attacker only can measure the runtime, this is a pretty good strategy to prevent timing attacks.

CHAPTER 4

A new solution using multiple threads

We will devise a new strategy to help prevent timing attacks. First we will demonstrate a timing attack, which will then be prevented by the proposed solution. We will also integrate the solution with a library implementing AES to demonstrate performance of the solution.

4.1 A simple timing attack example

We consider the vulnerable strcmp function mentioned earlier and perform an attack on it. The setup is where 1 string is a secret value, and the other is the input. The attacker times the runtime, and uses it to figure out the secret. Here is the victim function

```c
int string_cmp_victim(char* s1, char* s2) {
    int n1 = strlen(s1);
    int n2 = strlen(s2);
    if (n1 != n2)
        return 0;

    for (int i = 0; i < n1; i++) {
        if (s1[i] != s2[i])
            return 0;
    }
    return 1;
}
```

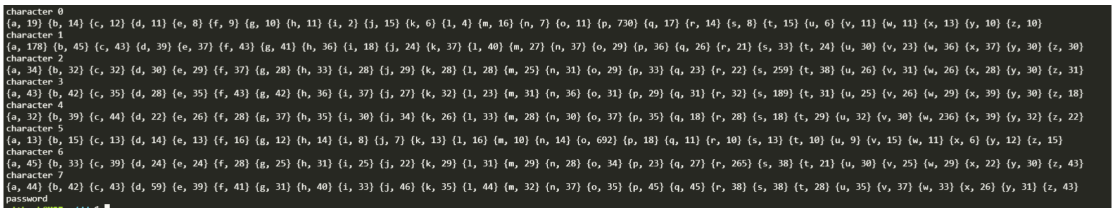
Here we see the early return if 2 characters don't match. So theoretically, it is possible to figure out a string by repeatedly calling this function with guesses of the other string.

Let us assume s1 = "password" which we want to guess using s2. For simplicity, let's assume we know the size of the string.

We start our guess as s2 = “aaaaaaaa”, and then time the function. Then we change only the first character, and try again, i.e. s2 = “baaaaaaa”. Try all possible values for the first spot, and the maximum time will be taken only by the correct guess. Repeat this by fixing the first i characters to figure out the (i+1)th character.

Does this idea work in practice? So let’s try it out… Below is an implementation of the above attack. To add more confidence to the guess, and to eliminate randomness due to setup/internal reasons, we make guesses based on the total time taken by 1000 runs of the function on the same input, and take the mode of 1000 guesses. (These numbers were tweaked based on experiments).

```c
char* password = "password";
char guess[8] = "aaaaaaaa";
for (int i = 0; i < 8; i++) {
    int counts[26] = {0};
    for (int l = 0; l < 1000; l++){
        long max_time = 0;
        char best_guess = 0;
        for (char j = 'a'; j <= 'z'; j++) {
            guess[i] = j;
            clock_t t = clock();
            for (int k = 0; k < 1000; k++)
                string_cmp_victim(guess, password);
            t = clock() - t;
            if (t > max_time) {
                max_time = t;
                best_guess = j;
            }
        }
        counts[best_guess-'a']++;
    }
    int max_count = 0;
    printf("character %d\n", i);
    for (int j = 0; j < 26; j++) {
        printf("{%c, %d} ", j+'a', counts[j]);
        if (counts[j] > max_count) {
            guess[i] = j + 'a';
            max_count = counts[j];
        }
    }
```



```
character 0
{a, 19} {b, 14} {c, 12} {d, 11} {e, 8} {f, 9} {g, 10} {h, 11} {i, 2} {j, 15} {k, 6} {l, 4} {m, 16} {n, 7} {o, 11} {p, 730} {q, 17} {r, 14} {s, 8} {t, 15} {u, 6} {v, 11} {w, 11} {x, 13} {y, 10} {z, 10}
character 1
{a, 178} {b, 45} {c, 43} {d, 39} {e, 37} {f, 43} {g, 41} {h, 36} {i, 18} {j, 24} {k, 37} {l, 40} {m, 27} {n, 37} {o, 29} {p, 36} {q, 26} {r, 21} {s, 33} {t, 24} {u, 30} {v, 23} {w, 36} {x, 37} {y, 30} {z, 30}
character 2
{a, 34} {b, 32} {c, 32} {d, 30} {e, 29} {f, 37} {g, 28} {h, 33} {i, 28} {j, 29} {k, 28} {l, 28} {m, 25} {n, 31} {o, 29} {p, 33} {q, 23} {r, 22} {s, 259} {t, 38} {u, 26} {v, 31} {w, 26} {x, 28} {y, 30} {z, 31}
character 3
{a, 43} {b, 42} {c, 35} {d, 28} {e, 35} {f, 43} {g, 42} {h, 36} {i, 37} {j, 27} {k, 32} {l, 23} {m, 31} {n, 36} {o, 31} {p, 29} {q, 31} {r, 32} {s, 189} {t, 31} {u, 25} {v, 26} {w, 29} {x, 39} {y, 30} {z, 18}
character 4
{a, 32} {b, 39} {c, 44} {d, 22} {e, 26} {f, 28} {g, 37} {h, 35} {i, 30} {j, 34} {k, 26} {l, 33} {m, 28} {n, 30} {o, 37} {p, 35} {q, 18} {r, 28} {s, 18} {t, 29} {u, 32} {v, 30} {w, 236} {x, 39} {y, 32} {z, 22}
character 5
{a, 13} {b, 15} {c, 13} {d, 14} {e, 13} {f, 16} {g, 12} {h, 14} {i, 8} {j, 7} {k, 13} {l, 16} {m, 10} {n, 14} {o, 692} {p, 18} {q, 11} {r, 10} {s, 13} {t, 10} {u, 9} {v, 15} {w, 11} {x, 6} {y, 12} {z, 15}
character 6
{a, 45} {b, 33} {c, 39} {d, 24} {e, 24} {f, 28} {g, 25} {h, 31} {i, 25} {j, 22} {k, 29} {l, 31} {m, 29} {n, 28} {o, 34} {p, 23} {q, 27} {r, 265} {s, 38} {t, 21} {u, 30} {v, 25} {w, 29} {x, 22} {y, 30} {z, 43}
character 7
{a, 44} {b, 42} {c, 43} {d, 59} {e, 39} {f, 41} {g, 31} {h, 40} {i, 33} {j, 46} {k, 35} {l, 44} {m, 32} {n, 37} {o, 35} {p, 45} {q, 45} {r, 38} {s, 38} {t, 28} {u, 35} {v, 37} {w, 33} {x, 26} {y, 31} {z, 43}
password
```

Figure 4.1: Successful attack on the victim function

```
    printf("\n");
}
printf("%s\n", guess);
```

See Figure 4.1 for a screenshot of the output of the attack.

We see the script very reliably guessing the correct password, just by measuring runtime of the function calls. Now let's see how to prevent this (there are many good existing solutions out there).

4.2 Proposed solution using multi-threading

I would like to propose a simple, but elegant solution that can work against all timing attacks. The key idea is derived from ct-fuzz which uses 2 threads, and runs the same program in parallel where 1 thread uses a randomized input and other uses the given inputs, and if both threads do not finish at nearly the same time, it errors.

This idea can be extended, to have one thread mask the runtime of the actual program, and hence, the attacker will not be able to decipher the true runtime of the program.

To do this, consider two threads, thread1 and thread2. Thread1 runs the program with the correct inputs. Thread2 runs the same program repeatedly with randomized inputs until thread1 completes. Once thread1 completes its execution, it sets a shared mutex variable to true, which signals to thread2 that thread1 is done. Thread2 checks this shared variable after each run, before rerunning with random inputs. Thread1 and Thread2 are barrier synchronized, and will terminate together.

How does this help? Because thread2 is the decider for the total runtime always (thread2 terminates only when thread1 terminates, and also waits for its current execution to complete before terminating), The total runtime will be only dependent on the randomized inputs. Why is this secure? There is no way for the attacker to get the actual runtime of the program, unless they can tap into the hardware (to see state of variables). There are a few other good things this approach gives us.

1. Healthy intermixing in the cache, can lead to slightly longer executions, but denies information to the attacker. Because both threads run the same program, and use similar variables (like the sbox), the same cache will be used by both threads, and can result in some rows being removed unpredictably, or other rows being added unpredictably. Usually attackers rely on the knowledge that something is present in the cache, and in this execution, it is not possible to guarantee that knowledge.

2. Can the attacker use a side channel, like power consumption to determine the actual runtime? This will not be possible in a single core, multi threaded environment as the threads will be scheduled in a round robin fashion, and the power consumption by both threads will look nearly identical. Theoretically, it wouldn't be possible for the attacker to distinguish both threads' power just by looking at the power trace, and will not be able to figure out where the first thread completes.

3. To be double careful, in case you do want to protect the runtime better, the first thread can wait an arbitrary number of milliseconds before flagging the termination. This can be easily done, without even using a sleep(), by just doing similar dummy operations (like xor, bitwise comparisons, etc) as done in the algorithm. To make the power trace similar to the normal executions.

4. Runtime is quantized to be equal to a number of runs of the algorithm on random inputs. Say the expected runtime of the algorithm is T. Then the expected runtime of the modified algorithm would be O(nT) where n is the number of times thread2 executed the algorithm. In practice, it is very likely for n=2 to be the outcome if we assume nearly similar runtimes for the algorithm. The differences in runtime due to extra branches, cache hits/misses, etc would not theoretically exceed the total runtime of the algorithm itself (T).

5. This means that theoretically, the slowdown caused by the algorithm would be 2x. We shall test this in practice, because a few things could take longer than expected (like setting up threads, and checking shared mutex variables).

6. Because of this quantization, runtimes will now fall in buckets T spaces apart. It is also very likely that most executions would lie in the same bucket (As argued above). This would mean, with significant probability, most executions take the same amount of time to finish. And anything that takes longer will be at least a whole T time units away.

7. The proposed solution is language agnostic, and can be implemented with any algorithm with any number of parameters. The idea is also simple enough, that it can be implemented in libraries which need it. It is also possible to write this as a generic, which would take in a function, with a set of parameters, and implement the masking functionality.

Thus we see a lot of great properties which would make it hard for attackers to actually extract useful information from the runtime of the program.

4.3 Implementation

We consider the same victim function we had above, and we use this as a black box, and write an algorithm around it. It doesn't matter what this algorithm is, but as long as we have the algorithm, and its inputs, we can implement our algorithm. We first define 2 structures which we will use later on in the implementation.

```
struct RunningStatus {
    int complete;
    int result;
    pthread_mutex_t lock;
};

typedef struct RunningStatus RunningStatus;


struct Args {
    char *s1;
    char *s2;
    RunningStatus *status;
};

typedef struct Args Args;
```

Running status is used to communicate the actual result, and message the second thread that the first thread has completed. It has a lock to prevent both threads from editing it at once. The threads will first wait for the other thread to release the lock before accessing variables.

Args is a structure that depends on the algorithm. This has all the parameters for the algorithms. And it has the running status. Both threads will be supplied with the same args object, and they will use these parameters to execute.

Next we need a randomizer for the inputs. For our situation we need a random string. Depending on the algorithm, this function is going to be different (for example, we might generate a random 128-bit secret for AES).

```c
char* get_random_string(int size){
    char *s = (char*)(malloc(size*sizeof(char)));
    for(int i = 0; i < size; i++)
        s[i] = 'a' + (rand() % 26);
    return s;
}
```

This is a simple function that calls rand() and generates an input of required size. Size is important, because size actually determines the runtime. So we would like to mock the true runtime, and generate a random input.

Now, we code the thread2's behaviour. Let's call this the masking function, which masks the runtime of the algorithm.

```c
void* string_cmp_masker(void *args) {
    Args *t_args = args;
    int cnt = 0;
    do {
        if (cnt == 1) cnt = 2;
        char* m1 = get_random_string(strlen(t_args->s1));
        char* m2 = get_random_string(strlen(t_args->s2));
        int result = string_cmp_victim(m1, m2);
        if (cnt != 2) {
            pthread_mutex_lock(&t_args->status->lock);
            cnt = t_args->status->complete;
            pthread_mutex_unlock(&t_args->status->lock);
        }
    } while (cnt < 2);
    return NULL;
}
```

The function takes a structure which has the arguments. We generate 2 random strings with the same size as the input. We have a cnt variable to ensure we run the randomized input 1 extra time after the true thread has completed. This can be removed, but it prevents the situation where while waiting for the lock, the first thread actually finished, and thus the second thread completes nearly at the same time as the first thread. The loop ends when cnt == 2, which happens 1 iteration after the status is marked as complete by the first thread.

Now we implement the first thread's behaviour. This is easier to do.

```c
void* string_cmp_main(void *args) {
    Args *t_args = args;
    int result = string_cmp_victim(t_args->s1, t_args->s2);
    pthread_mutex_lock(&(t_args->status->lock));
    t_args->status->complete = 1;
    t_args->status->result = result;
    pthread_mutex_unlock(&t_args->status->lock);
    return NULL;
}
```

We just call the victim function, pass the true values, and then mark the status as complete, and update the result. We also use locks to prevent multiple updates or race conditions between the threads.

Now the final integrating function which sets up the 2 threads

```c
int string_cmp_fixed(char* s1, char* s2) {
    RunningStatus *status = (RunningStatus *)malloc(sizeof(
   RunningStatus));
    pthread_t thread1, thread2;
    status->complete = 0;
    status->result = -1;
    pthread_mutex_init(&status->lock, NULL);
    pthread_mutex_unlock(&status->lock);
    Args args1; args1.s1 = s1; args1.s2 = s2; args1.status = status;

    int ret1 = pthread_create(&thread1, NULL, string_cmp_main, (void
   *)&args1);
    int ret2 = pthread_create(&thread2, NULL, string_cmp_masker, (
   void*)&args1);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return status->result;
}
```

4.4 Implementing on cryptomaniac

We then go ahead and modify the code of cryptomaniac, which is a popular open source implementation of AES encryption in C. This was easy to understand and modify, and hence was chosen to modify.

The code changes follow the same structure as above, only the Args struct, and the randomizer will need to be changed.

Here is the full set of changes that were added. After that, we did a timing analysis to determine the performance hit on a real algorithm.

```

struct RunningStatus {
    int complete;
  int result;
    pthread_mutex_t lock;
};

typedef struct RunningStatus RunningStatus;

struct Args {
  char *infile;
  char *outfile;
  void *key;
  void *iv;
  EVP_CIPHER *cipher;
  int enc;
    RunningStatus *status;
};

typedef struct Args Args;


void* encrypt_main(void *args) {
    Args *t_args = args;
    int result = aes_encrypt_file(
    t_args->infile, t_args->outfile, t_args->key, t_args->iv,
    t_args->cipher, t_args->enc, 1);
    pthread_mutex_lock(&(t_args->status->lock));
    t_args->status->complete = 1;
  t_args->status->result = result;
    pthread_mutex_unlock(&t_args->status->lock);
    return NULL;
}

char* get_random_key(){
    char *s = (char*)(malloc(EVP_MAX_KEY_LENGTH*sizeof(char)));
  char *temp = "0123456789ABCDEF";
    for(int i = 0; i < EVP_MAX_KEY_LENGTH; i++) {
        s[i] = temp[rand() % 16];
  }
  char *res = (char*)(malloc(EVP_MAX_KEY_LENGTH*sizeof(char)));
    hex2bin(s, res, EVP_MAX_KEY_LENGTH);
  return res;
}

char* get_random_iv(){
    char *s = (char*)(malloc(EVP_MAX_IV_LENGTH*sizeof(char)));
  char *temp = "0123456789ABCDEF";
    for(int i = 0; i < EVP_MAX_IV_LENGTH; i++) {
        s[i] = temp[rand() % 16];
  }
  char *res = (char*)(malloc(EVP_MAX_IV_LENGTH*sizeof(char)));
    hex2bin(s, res, EVP_MAX_IV_LENGTH);
  return res;
}

void* encrypt_masker(void *args) {
    Args *t_args = args;
    int cnt = 0;
    do {
        if (cnt == 1) cnt = 2;
        char* key = get_random_key();
        char* iv = get_random_iv();
        int result = aes_encrypt_file(
      t_args->infile, t_args->outfile, key, iv,
      t_args->cipher, t_args->enc, 0);
        if (cnt != 2) {
        cnt = t_args->status->complete;
```

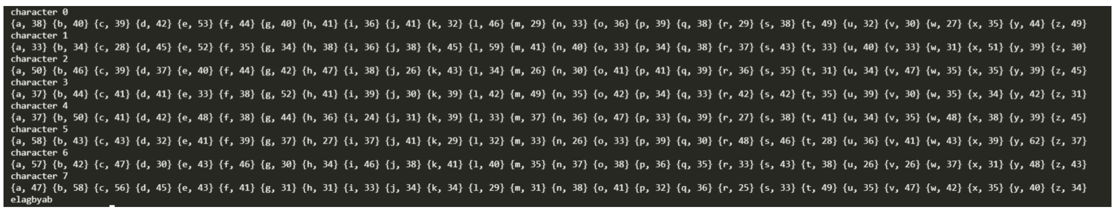
```c
        }
    } while (cnt < 1);
    return NULL;
}


int encrypt_fixed(const char * infile, const char * outfile, const
   void * key, const void * iv, const EVP_CIPHER * cipher, int enc) {
    RunningStatus *status = (RunningStatus *)malloc(sizeof(
   RunningStatus));
    pthread_t thread1, thread2;
    status->complete = 0;
  status->result = -1;
    pthread_mutex_init(&status->lock, NULL);
    pthread_mutex_unlock(&status->lock);
    Args args1;
  args1.infile = infile; args1.outfile = outfile;
  args1.key = key; args1.iv = iv;
  args1.cipher = cipher; args1.enc = enc;
  args1.status = status;

    pthread_create(&thread1, NULL, encrypt_main, (void*)&args1);
    pthread_create(&thread2, NULL, encrypt_masker, (void*)&args1);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
  return status->result;
}
```

We use 2 randomizers, 1 to get a random Key, and 1 to get a random Initialization Vector. We generate random hex values and then convert them to binary.

4.5 Results

We first look at the correctness by rerunning the same above attack on the modified strcmp function which uses the masker.

We see in Figure 4.2 that all the characters are predicted at nearly equal counts (and



```
character 0
{a, 38} {b, 40} {c, 39} {d, 42} {e, 53} {f, 44} {g, 40} {h, 41} {i, 36} {j, 41} {k, 32} {l, 46} {m, 29} {n, 33} {o, 36} {p, 39} {q, 38} {r, 29} {s, 38} {t, 49} {u, 32} {v, 30} {w, 27} {x, 35} {y, 44} {z, 49}
character 1
{a, 33} {b, 34} {c, 28} {d, 45} {e, 52} {f, 35} {g, 34} {h, 38} {i, 36} {j, 38} {k, 45} {l, 59} {m, 41} {n, 40} {o, 33} {p, 34} {q, 38} {r, 37} {s, 43} {t, 33} {u, 40} {v, 33} {w, 31} {x, 51} {y, 39} {z, 30}
character 2
{a, 50} {b, 46} {c, 39} {d, 37} {e, 40} {f, 44} {g, 42} {h, 47} {i, 38} {j, 26} {k, 43} {l, 34} {m, 26} {n, 30} {o, 41} {p, 41} {q, 39} {r, 36} {s, 35} {t, 31} {u, 34} {v, 47} {w, 35} {x, 35} {y, 39} {z, 45}
character 3
{a, 37} {b, 44} {c, 41} {d, 41} {e, 33} {f, 38} {g, 52} {h, 41} {i, 39} {j, 30} {k, 39} {l, 42} {m, 49} {n, 35} {o, 42} {p, 34} {q, 33} {r, 42} {s, 42} {t, 35} {u, 39} {v, 30} {w, 35} {x, 34} {y, 42} {z, 31}
character 4
{a, 37} {b, 50} {c, 41} {d, 42} {e, 48} {f, 38} {g, 44} {h, 36} {i, 24} {j, 31} {k, 39} {l, 33} {m, 37} {n, 36} {o, 47} {p, 33} {q, 39} {r, 27} {s, 38} {t, 41} {u, 34} {v, 35} {w, 48} {x, 38} {y, 39} {z, 45}
character 5
{a, 58} {b, 43} {c, 43} {d, 32} {e, 41} {f, 39} {g, 37} {h, 27} {i, 37} {j, 41} {k, 29} {l, 32} {m, 33} {n, 26} {o, 33} {p, 39} {q, 30} {r, 48} {s, 46} {t, 28} {u, 36} {v, 41} {w, 43} {x, 39} {y, 62} {z, 37}
character 6
{a, 57} {b, 42} {c, 47} {d, 30} {e, 43} {f, 46} {g, 30} {h, 34} {i, 46} {j, 38} {k, 41} {l, 40} {m, 35} {n, 37} {o, 38} {p, 36} {q, 35} {r, 33} {s, 43} {t, 38} {u, 26} {v, 26} {w, 37} {x, 31} {y, 48} {z, 43}
character 7
{a, 47} {b, 58} {c, 56} {d, 45} {e, 43} {f, 41} {g, 31} {h, 31} {i, 33} {j, 34} {k, 34} {l, 29} {m, 31} {n, 38} {o, 41} {p, 32} {q, 36} {r, 25} {s, 33} {t, 49} {u, 35} {v, 47} {w, 42} {x, 35} {y, 40} {z, 34}
elagbyab
```

Figure 4.2: Successfully prevented attack on the victim function

that the output is actual gibberish).

Thus we see that this actually does confuse the attacker, and makes it impossible to perform a timing attack based on the runtime.

Now we will do a performance analysis using some crypto algorithms. For this, we use this repo which provides an implementation to encrypt a file using AES-256 with OpenSSL's library.

We modify the above implementation based on our algorithm, and time the run. Here are the results on an input file which was 90Mb in size.

Table 4.1: Performance effect on different modes of operation supported by cryptomaniac

Situation	Time (raw algo) in s	Time (modified algo) in s	Slowdown
Encrypt CBC	0.17	0.378	2.22x
Decrypt CBC	0.112	0.215	1.91x
Encrypt block mode	0.092	0.216	2.34x
Decrypt block mode	0.129	0.241	1.86x

We see that the modified algorithm results in around 2x slowdown as predicted theoretically above.

4.6 Further possibilities

Instead of immediately terminating when thread1 completes, thread2 can take a decision with some probability p on whether to terminate or not. This can further add noise to the masked runtimes, which would make it impossible to use the runtime to predict anything about the actual runtime. We could use an array of threads, instead of a pair, but not sure what value this provides

CHAPTER 5

Detect Meltdown style attacks using static analysis

In this chapter we will look at an initial attempt to look solve memory leaks like meltdown using a static analysis pass. This idea was proposed during the project but not pursued as we felt meltdown had good existing solutions and was nearly a solved problem. However, if we manage to ensure security with a static pass that would mean this can be caught during compile time, and the program can run faster without the runtime safety guards that were added as prevention for meltdown.

5.1 Attack scenarios considered

Typically I would categorize micro-architectural attacks into 2 main categories. Attacks that use just 1 program/thread (like meltdown), where the attacker exploits some internal behavior failure to extract data. The others are situations where there is 1 victim program and 1 attacker program. This include scenarios like the flush + reload, prime + probe attack, etc which were discussed in Chapter 1. The previous chapters aimed at fixing multi-process attacks, while in this chapter we will explore a static analysis based technique that can possibly prevent a single process attack.

5.2 Attack methodology

The attacks of this type typically rely on an out of bounds memory access, through speculative execution, which gets cached. Then the attacker uses some creative method to actually extract the cached data. As an example, we have meltdown which was described in Chapter 1 where the attacker would cache a block in the cache and then look at the timing profile to determine which block was cached.

The only way the attack works, and exposes some internal data is if the program manages to violate memory safety in a speculative sense. Assuming we can ensure that all memory accesses are safe, in a speculative sense, the attacker will not be able to read outside the scope of the program. As this is a single process attack, and the attacker controls the attack program, it would make the attacks pointless because the attacker owns the program that he is able to read data from.

5.3 A minimal example

We consider a small example that we can use to test out a proof of concept solution. We assume that the program contains only load or store instructions, and we use only integers, no arrays. Pointers are allowed.

An invalid access can happen if you load from an out of bounds address. Dereferencing pointers to valid integers is a valid operation, while dereferencing an arbitrary pointer is not valid.

Keeping these constraints in mind, lets define some book-keeping we would like to do to be able to code up these checks.

5.4 A variable's state

We define a "state" of a variable. Each operation in the program modifies the state of the variables involved. Lets consider our minimal example above, and our variable state would need 2 things. First being the range of values it can take. To represent the range, we will create a Range data structure. Secondly, for pointers, it doesn't make sense to know the range as it would store a variable's address (if valid obviously). So we would like to know the variable it points to. This is the second attribute of the state.

5.5 The range data structure

To effectively store and process ranges, we need to create a range data structure, which supports all arithmetic operations. A range is defined as a collection of subranges. We create a subrange using 2 numbers which denote the start and end of the subrange, and 2 booleans denoting if the endpoints are included or not.

To define a range, we create 4 sets.

1. Set of included discrete points.
2. Set of all excluded points (e.g if we do "if(x!=0)" then the state of x will exclude 0).
3. Set of all included subranges. These will be validated to be non-overlapping.
4. Set of all excluded subranges. These will also be validated to be non-overlapping.

The range data structure should support addition/deletion of of a point, while preserving validity of the range. There will be utility functions that process 2 ranges under some arithmetic operation, and give the output range.

5.6 Solving the minimal example

Now assuming each variable has a state, we go ahead and do the following.At each load instruction, we update the state of the variable being loaded to if the right hand side is a $constantExpr$. This can be implemented by figuring out the type of the operand in LLVM. If it isn't a constant expression, then it is a pointer dereference, which needs to be checked for safety. There are 2 possibilities, the pointer points to a variable, in which case it is safe, or the pointer stores a constant value, in which case it probably isn't safe. We check the state of the pointer. If the variable it points to is null, then we deem it unsafe. If it points to a known variable, then it is considered safe. This ensures memory safety in this small scenario.

5.7 Generalizing from the minimal example

5.7.1 Arithmetic operations

These can be handled by processing the states properly using the range manipulation utilities that will be provided. There are no other special considerations that need to be added.

5.7.2 Arrays

Now each pointer will need another state attribute. We need to know the size of the array it points to, if it is an array. This is because the safe range for a pointer is larger. This new state attribute needs to be managed for pointers. The size can be obtained from the LLVM at the initialization statement.

5.7.3 Branching instruction

This is the most important instruction because all the attacks rely on this. This is where specultative execution could change the course of execution. These can be of 2 types. In the control flow graph, if there is a backward edge due to the branch, it is a loop. If there is a forward edge then it is a conditional statement.

To handle these types of instructions, we define a speculative execution limit. This is the max number of instructions that can be executed speculatively. So now, each variable can have a set of possible states in a particular spot in the program depending on the paths it could have taken. If all states are safe, the program is deemed safe.

At every branch, we consider both possibilities (branch or no branch) unless it is an unconditional jump. This will ensure that the program is safe under a speculative setting.

Let us consider cases where the loops execute a finite known number of times. This is the case for most cryptographic algorithms. If the loop executes for an unknown or a user driven number of iterations, it will not be possible to figure out when to terminate the analysis. This can be considered in future work.

5.7.4 Function calls

Lets consider calls to user-defined functions. Each function can be analysed, and we can compute a "Read set" for the function. The "Read set" would depend on the function parameters, any global variables and constants (dependence on internal variables can be replaced with some expressions of the above, or constants). Each element in the "Read set" should specify the operations to be performed on the inputs before performing the read. It should also specify (if any) conditions need to be met before checking for safety. For each function, we also store an “Output set” which again depends on the same inputs as above. It will also store conditions, based on which a particular output is chosen.

Functions will be analysed as follows. First we obtain the call graph of the module. The call graph will have a relation such that if there is an edge (f, g) then f calls g. This would be a rooted dag, rooted at main(). Any cycles would mean a cyclic call, which is a compile issue, so hence cycles can’t be present, making it a dag. By traversing this graph, we can compute the read sets and outputs for every node. We can perform a recursive dfs, and do the computation when we are going to return to the parent (at that point, all its children will have been computed already, and its results can be used in the computation at the current node, as it calls its children). After the DFS, the full module would be analysed.

If the code uses library functions, we would need to analyse those too. If its possible to generate a single .ll file with all functions called, then it can be analysed in 1 pass. If not, there needs to be a way to store the read and output sets of all functions for the libraries. Once we can generate that, we can load that for the library function in question and use that in our analysis. This can be considered for future work in this area.

5.8 Conclusion

With some constraints, it will be possible to ensure memory safety in a speculative fashion with just static analysis. We did implement a small pass that solved programs within the "minimal example" constraints. The generalizations were planned out in theory, and are expected to provide the core ideas to be able to solve the memory bound checking problem.

This can then be integrated in the compilation to prevent code with such vulnerabilities to compile successfully. Thus, attackers will not then be able to execute programs which exploit the memory out of bounds to extract information they dont have access to.

REFERENCES

1. **He, S.**, **M. Emmi**, and **G. Ciocarlie**, ct-fuzz: Fuzzing for timing leaks. *In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020.

2. **Kocher, P.**, **J. Horn**, **A. Fogh**, , **D. Genkin**, **D. Gruss**, **W. Haas**, **M. Hamburg**, **M. Lipp**, **S. Mangard**, **T. Prescher**, **M. Schwarz**, and **Y. Yarom**, Spectre attacks: Exploiting speculative execution. *In 40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

3. **Lattner, C.** and **V. Adve**, Llvm: A compilation framework for lifelong program analysis amp; transformation. *In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04. IEEE Computer Society, USA, 2004. ISBN 0769521029.

4. **Lipp, M.**, **M. Schwarz**, **D. Gruss**, **T. Prescher**, **W. Haas**, **A. Fogh**, **J. Horn**, **S. Mangard**, **P. Kocher**, **D. Genkin**, **Y. Yarom**, and **M. Hamburg**, Meltdown: Reading kernel memory from user space. *In 27th USENIX Security Symposium (USENIX Security 18)*. 2018.

5. **Schinzel, S.**, An efficient mitigation method for timing side channels on the web. 2011.

6. **Wikipedia contributors** (2021). Timing attack — Wikipedia, the free encyclopedia. URL `https://en.wikipedia.org/w/index.php?title=Timing_attack&oldid=1026700778`. [Online; accessed 18-June-2021].

7. **Wu, M.**, **S. Guo**, **P. Schaumont**, and **C. 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. Association for Computing Machinery, New York, NY, USA, 2018. ISBN 9781450356992. URL `https://doi.org/10.1145/3213846.3213851`.