Towards Cycle-Level Verification of Constant-Time Cryptography

by

Jessica Y. Xu

S.B. in Computer Science and Engineering, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Jessica Y. Xu Department of Electrical Engineering and Computer Science May 28, 2024
Certified by:	Anish Athalye Doctoral Candidate, Thesis Supervisor
Certified by:	Nickolai Zeldovich Professor, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Towards Cycle-Level Verification of Constant-Time Cryptography

by

Jessica Y. Xu

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Cryptographic primitives–hash functions, symmetric key encryption algorithms, asymmetric key exchange algorithms, and more–are used everywhere to achieve security in modern computing. Since these algorithms have complicated, math-heavy implementations, they are typically used through cryptographic library functions. However, many timing side-channel attacks, which leak information when execution time depends on secrets, have been found in popular cryptographic libraries, such as OpenSSL. Formal verification aims to rule out timing side channels in cryptographic software.

This thesis presents Quake, a framework for verifying cryptographic library functions are constant-time for a specific hardware implementation, regardless of where the code is located in memory. Quake represents the location of code in memory using symbolic addresses and introduces a ROM model that gets concrete memory data from symbolic addresses. This thesis evaluates Quake and demonstrates that it can detect address-dependent timing behavior and does so in a reasonable amount of time.

Thesis supervisor: Anish Athalye
Title: Doctoral Candidate

Thesis supervisor: Nickolai Zeldovich
Title: Professor

Acknowledgments

Thank you to Anish, for being an incredible mentor and role model over the last year and a half. He was extremely generous with his time and always eager and willing to help. This thesis would not have been possible without his guidance, expertise and support. Thank you to Professor Nickolai Zeldovich for his advice and feedback.

Thank you to my friends, for the memories, laughs, and support through all the ups and downs of the last five years. My MIT experience would not have been the same without them. Special shoutout to sMITe, past and present, for bringing me so much joy and for being such a core part of my time here.

Thank you to my brother, for being an incredible role model throughout my life and for all the advice he's ever given me. And finally, thank you to my parents for their unconditional love, support, and encouragement.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Cryptographic primitives are used everywhere to achieve security in modern computing. Cryptographic hash functions like SHA-256 are used for file authentication; symmetric key algorithms like AES are used for encrypting data with a shared secret; and asymmetric key encryption algorithms such as RSA, the Diffie-Hellman key exchange protocol, and elliptic-curve cryptography algorithms are used for digital signatures and sharing secrets. These algorithms are critical to many cryptographic protocols. For example, Transport Layer Security (TLS) is the protocol for network security used in email, instant messaging, and most notably, HTTPS, the protocol used for communication over the World Wide Web. TLS uses both asymmetric and symmetric keys: an asymmetric key exchange algorithm establishes the shared secret session key, which is the symmetric key used to encrypt network traffic for the remainder of the session.

These cryptographic primitives are nontrivial and often rely on complicated math, so their implementations are best left to better-tested and used cryptography libraries, such as OpenSSL, NaCl, and HACL* [1]. Since these libraries are part of the trusted computing base for so many applications, it's critical that their implementations are secure. One class

of attacks we are concerned about is timing side-channel attacks. Timing side-channels leak information when execution time depends on secrets. Attacks may take advantage of speculative execution for branch instructions that depend on secrets, non-constant-time operations such as integer division, or cache timing.

Vulnerabilities are frequently found in popular cryptographic software libraries. For example, many timing attacks have been found for OpenSSL, the most widely used open-source TLS library. Ammanaghatta et. al. [2] found a bug in an OpenSSL function that accessed two cache lines in different orders depending on the value of a secret offset. Brumley and Boneh [3] demonstrate that attacks against network servers are practical by developing a remote timing attack against OpenSSL-based web servers. Their attack extracts the private key stored on the server by measuring the time for the OpenSSL server to respond to decryption queries. Brumley and Tuveri [4] devise another timing attack on OpenSSL that recovers the private key from a TLS server using a timing vulnerability in the scalar multiplication implementation for elliptic curves. These timing side-channels are a major threat to security, so it is important that cryptographic code is constant-time. Since timing attacks buried in complex implementation details are hard to find and eliminate, we turn towards formal verification to rule out timing side channels.

The standard approach to verifying the timing behavior of cryptographic software assumes a leakage model and proves that software is constant-time with respect to a specific leakage model [5]–[7]. However, there can be a gap between leakage models and the hardware's behavior, especially as there are many different leakage models and processors. For example, leakage models that assume the program counter address and memory-access addresses are leaked do not account for non-constant-time multiplication in processors such as the ARM Cortex-M3 [8]. Wang et. al. attempt to bridge this gap with a tool to verify processors against leakage contracts [9].

Quake takes a different approach of directly verifying software against hardware. It uses symbolic execution of circuits to verify that a piece of software runs in a constant number

of cycles on a particular processor.

Since the specific piece of software we want to verify is a *library function* that is one part of a larger program, we need to consider the context in which that function is run. For example, there may be existing register values in the processor, data on the stack, or other instructions in the processor's pipeline. And since we do not know the address a library function will be linked to at verification time, we should consider all possible locations for code as well.

Quake aims to verify that the timing behavior of code does not depend on where the code is located in memory by executing it using a *symbolic* instruction address. Quake only verifies software starting from a reset processor state and does not reason about how changing the initial processor state may affect timing behavior, which would be necessary to achieve true cycle-level verification of constant-time cryptography.

1.2 Goals and Challenges

The security goal of this work is to verify that cryptographic library functions are constant-time at a cycle level regardless of the memory address the code is located. Since we don't know the context in which cryptographic library code is run, and in particular, the address of the library function, verification attempts that assume code is located at a specific memory address are not complete.

Our verification approach simulates software running on a system-on-a-chip using symbolic execution. Symbolic execution allows for both concrete values and symbolic values, which represent all possible values of a particular datatype. Our simulation runs on circuit models that have both symbolic and concrete state elements.

Designing Quake to verify code located at an unknown address requires reasoning about executions where the instruction addresses are unknown. In order to simulate the program execution, we need to derive the correct instruction data from an unknown instruction ad-

dress. We represent unknown instruction addresses with symbolic values, but a key challenge we solve is how to read the correct data from those symbolic addresses.

We represent these unknown instruction addresses in Quake with symbolic values. Besides the instruction data, many other circuit state elements depend on the symbolic instruction address. As operations are executed over the course of the simulation, the symbolic expressions for these circuit state elements may grow in size. Symbolic execution proves statements about expressions using an SMT solver. However, since the SMT problem is NP-hard, large symbolic expressions could make verification intractable. As a result, a second key challenge in designing Quake is ensuring symbolic expressions in our state do not grow too large.

1.3 Contributions

This thesis makes the following contributions:

1. An approach for symbolic circuit execution where code is located at an unknown instruction addresses. This approach introduces a *ROM model* that determines the instruction from a symbolic instruction address

2. Quake, a verification framework that verifies cryptographic library functions are constant-time at a cycle level regardless of the memory address the code is located. Quake includes an implementation of the ROM model for a specific processor

3. An evaluation of Quake, which demonstrates that Quake is able to detect timing variations that depend on the instruction address code is located. It also examines how verification time is affected by whether we have a symbolic or concrete instruction address.

1.4 Thesis Outline

The rest of this thesis is outlined as follows. Chapter 2 describes relevant background. Chapter 3 describes the approach used by Quake to verify software is constant-time regardless of where the code regardless of where code is located. Chapter 4 describes the developer interface to the Quake framework. Chapter 5 describes how we implement Quake for the Ibex processor. Chapter 6 describes some performance enhancements we tried. Chapter 7 evaluates Quake on whether it can detect address-dependent timing variations and how verification time is affected by whether we have concrete or symbolic instruction addresses. Chapter 8 discusses limitations and areas for future work. Finally, Chapter 9 details related work and Chapter 10 concludes the thesis.

Chapter 2

Background

2.1 Symbolic Execution of Circuits

The verification technique used in Quake is based on symbolic execution. Symbolic execution allows us to efficiently reason about many possible states of a program's execution.

Symbolic execution computes not only on concrete values, but also *symbolic values*, which can represent all possible values of a particular datatype. For example, a symbolic 3-bit value `x` represents the values 0, 1, 2, ... 7 at the same time. *Symbolic expressions* are mathematical expressions based on symbolic values. For example, `x ≫ 1` represents 0, 1, 2, or 3.

During symbolic execution of a program, symbolic values are initialized to represent all possible values. When there's a branching condition, the execution creates a path condition with the if constraint for the if branch and the else constraint for the else branch. It can then use an SMT solver, such as Z3 [10], to prove statements about expressions, such as finding concrete values for expressions or proving paths unreachable.

Since Quake aims to do cycle-level simulations of processors running some code, we would like to do symbolic execution with circuits. Quake uses the Verilog-to-Rosette toolchain from rtlv [11], which compiles Verilog circuits to programs in Rosette, a domain-specific language for solver-aided programming built on top of Racket. This toolchain first uses the Yosys [12]

synthesis tool to generate an SMT-LIB representation of the circuit. Then a domain-specific language provided by rtlv called `#lang yosys` transforms the Yosys output into Rosette code.

This pipeline gives us a Rosette struct that represents the circuit state, including register fields, memory, and current input values. It also gives a step function that accepts a circuit state and returns the new state given by running the circuit for one clock cycle. And finally, it gives us functions for setting inputs, getting outputs, and updating elements of the circuit state. As a result, we get a Rosette state machine of the Verilog circuit that allows us to execute the circuit entirely with Rosette and take advantage of Rosette's support for solver-aided programming.

Furthermore, rtlv has some utility functions that are helpful for simplifying symbolic expressions in our circuit state. We use the `concretize` function, which uses the SMT solver to attempt to evaluate a symbolic expression to a single concrete value. It returns the concrete value if successful or the original symbolic expression otherwise. We also use the `overapproximate` function, which returns a fresh symbolic.

Quake uses the representation of circuit state as concrete and symbolic values to verify that software located at any instruction address runs in a constant number of cycles on a particular processor. The instruction address where code is located is initialized as a symbolic value. Symbolic execution of the code shows that the software runs in a constant number of cycles by allowing us to efficiently check every possible memory address for code.

2.2 Position Independent Code

Since our goal is to verify cryptographic library functions that are one part of a larger program, we need to account for the fact that at verification time, we do not know where the code will be located.

One approach might be to have the static linker load code at a symbolic address. However,

it is unclear how to do this as instructions would need to contain symbolic values as well.

Instead, Quake compiles code to be position-independent and represents the base address of that code as a symbolic value. Position-independent code (PIC) is code that will execute correctly regardless of its absolute addresses. Code that is not position-independent may depend on absolute addresses, meaning it cannot be located at any address. We need to be able to model code that can be located at any address to represent library functions that may be loaded at arbitrary addresses.

Thus, Quake needs the software it verifies to be compiled to PIC to verify that its timing behavior does not depend on the address code is located.

Chapter 3

Approach

This chapter presents Quake, a verification framework for verifying the timing behavior of software run on a particular processor is independent of the location of the code in memory. Since we aim to verify timing behavior of software on a hardware implementation without the abstraction of a leakage contract, Quake runs concrete programs from reset state until termination and counts the number of hardware cycles. In order to more efficiently reason about arbitrary processor states, Quake uses symbolic variables to represent the processor state and simulate running the program with symbolic execution. And finally, to verify that timing behavior does not depend on the location of code in memory, Quake uses symbolic memory addresses for the ROM. Quake introduces an approach to model these symbolic memory addresses with a *ROM model* that allows us to read concrete data from a symbolic address.

3.1 Software on Symbolic Hardware

Quake directly verifies software against a hardware *implementation* rather than a model of the hardware. It reasons about a complete program with cryptographic code, like Listing 3.1, which corresponds to the software input to the framework in Figure 3.1.

Tools like Icarus Verilog and Verilator do cycle-accurate simulations of processors running

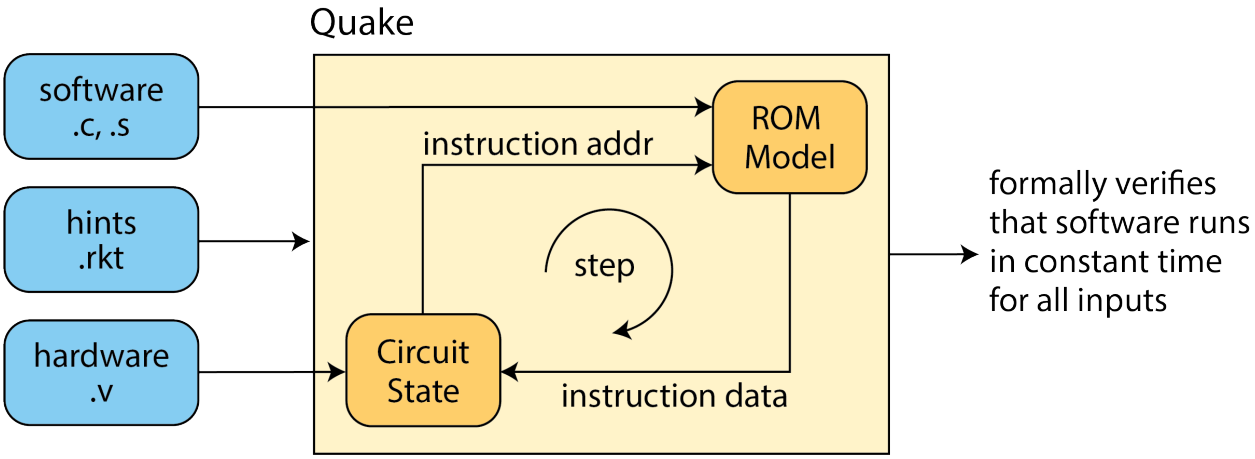

Figure 3.1: System diagram of verification approach for symbolic instruction addresses. The boxes in blue are user inputs to the framework

```
#include "ed25519.h"

#define MSG_SIZE 100

void main() {
    uint8_t pk[32], sk[64], msg[MSG_SIZE], sig[64];
    ed25519_sign(sig, msg, MSG_SIZE, pk, sk);
}
```

Listing 3.1: Example program passed in to Quake that calls a cryptographic function

some code, but these simulators run on concrete values: every bit is a 0 or a 1. We could run a simulation for every possible secret key value, but with 2^{32} possible values for a 32-bit key, this brute-force approach is not practical.

Instead, Quake uses a symbolic hardware simulator to more efficiently reason about processors executing cryptographic code. A symbolic simulator allows us to mark certain circuit state elements, such as those corresponding to `sk[]` and `pk[]` in our example, as symbolic variables, and then symbolically execute the circuit to determine whether execution time depends on these variables.

Symbolic simulation allows us to reason about the number of cycles a circuit at any symbolic starting state takes to reach a state satisfying a property. Quake stops simulation when `main()` returns, which allows it to reason about the number of cycles needed to execute an entire program from start to end. Quake starts the processor from a reset state, so circuit state elements that have concrete values in the reset state are initialized with those concrete values. As a result, the symbolic simulator allows us to reason about the number of cycles a circuit starting from a reset state takes to finish executing a program with cryptographic code.

To create the hardware simulator, Quake requires developers to pass in a complete hardware implementation, as seen in Figure 3.1.

3.2 ROM Model for Symbolic Memory Addresses

The symbolic circuit state model in Quake uses symbolic values to represent concrete memory locations, which allows it to reason about all possible values for a particular circuit state element. However, this thesis aims to reason about code located at arbitrary memory addresses and read concrete values from those memory addresses. To do this, Quake uses symbolic memory addresses in its circuit state and introduces a *ROM model*, as seen in Figure 3.1, that determines the concrete instruction data from a symbolic instruction address.

```
base_addr = new_symbolic()
program = []

def initialize(instr):
  program = instr

  circuit_state = new_circuit()
  return circuit_state.update(base_addr)

def step(circuit_state):
  updated_circuit = read_rom_model(circuit_state)
  ...

def read_rom_model(circuit_state):
  offset = concretize(circuit_state.instr_addr - base_addr)
  instr_addr = base_addr + offset
  instr_data = program[offset]
  return circuit_state.update(instr_data)
```

Listing 3.2: Pseudocode that shows how the ROM model is used to read concrete instruction data from symbolic instruction addresses

Quake uses a symbolic base address for the location of code in memory. At the start of verification, it creates a new symbolic circuit state that is initialized with a symbolic base address. Quake stores this base address and the software instruction data in the framework, as shown in the declaration of global variables and the `initialize` function in Listing 3.2. Then, at each cycle of program execution, it uses the ROM model to update the circuit state with the concrete instruction data from the symbolic instruction address, as shown in the `read_rom_model` function in Listing 3.2. The ROM model reads the symbolic instruction address from the circuit state and computes the offset of this symbolic instruction address from the stored symbolic base address. It uses the `concretize` utility function to evaluate this offset to a single concrete value. This concrete offset is used to index into the stored program code to get the correct data. Finally, the ROM model updates the circuit state with the correct instruction data, so that it can be used in the next cycle of execution.

3.3 ROM Model-Aware Circuit State Transformations

Since different processors have different instruction and data interfaces, Quake implements the ROM model for a specific processor. Depending on how the chosen processor works, other circuit state elements may also depend on the instruction address. For example, Quake uses the Ibex processor [13], which has an instruction prefetch buffer for performance that stores the instructions themselves with the instruction address they came from.

Since Quake uses symbolic instruction addresses and other circuit state elements may depend on the instruction address, the size of symbolic expressions in the state grows as operations are executed. The SMT solver used to prove statements about symbolic expressions solves an NP-hard problem, so queries become intractable as expressions become large. To avoid timing out during circuit simulation, Quake performs various circuit state transformations that simplify symbolic expressions.

For the Ibex processor, Quake uses three approaches to simplify symbolic expressions in the circuit state that depend on the symbolic base address.

1. Circuit state elements that can be evaluated to one concrete value are replaced with that concrete value.

2. 32-bit circuit state elements which are a concrete offset from the base address are replaced with the base address plus that concrete offset

3. 31-bit circuit state elements which are a concrete offset from the 31-bit prefix of the base address are replaced with the 31-bit prefix of the base address plus that concrete offset

These transformations were sufficient for making verification finish in a reasonable amount of time. However, they are specific to how the Ibex processor uses the instruction address, so may not apply to other processors.

3.4 Performance Hint Interface

Quake has a performance hint interface that allows for developers to provide hints on how to transform the circuit state during symbolic execution. Quake includes a few types of hints that developers can use how they wish. Since developers are providing the hints, the hints themselves are untrusted. The system is responsible for ensuring that incorrect hints do not lead to circuit state transformations that reduce the space of possible circuit states, as that may lead to missing executions with variable timing behavior, causing our system to incorrectly verify that a piece of software runs in constant time.

One key hint in verifying timing behavior is the `case-split` hint, which Quake applies when there is a branching instruction. The `case-split` hint executes each case separately, applying the path condition associated with that case in all future cycles of the simulation. This hint allows us to execute branches separately and compare their timing behavior at the end to verify whether all branches run in the same number of hardware cycles.

Another key hint that is particularly relevant for executing with symbolic memory addresses is the `concretize-with-function` hint. This hint allows for developers to pass in their own function that is called with the circuit state to update it, but Quake ensures the function does not reduce the space of possible circuit states. As described above, executing with symbolic memory addresses is processor-specific as different processors have different memory interfaces. Since the hint interface is for general use, Quake should not include a hint that supports processor-specific symbolic memory address transformations. Instead `concretize-with-function` allows developers to pass in functions that do more complex transformations to circuit state for execution with symbolic memory addresses.

Chapter 4

Developer Interface

This chapter describes the user interface to the Quake verification framework. Since Quake verifies cryptographic software against a specific hardware implementation, users supply the software and hardware. They also provide performance hints as necessary to help with verification.

4.1 Hardware

Users supply a hardware implementation in the form of a Verilog system-on-a-chip. This circuit is compiled to a Rosette state machine representation of the circuit, which allows for

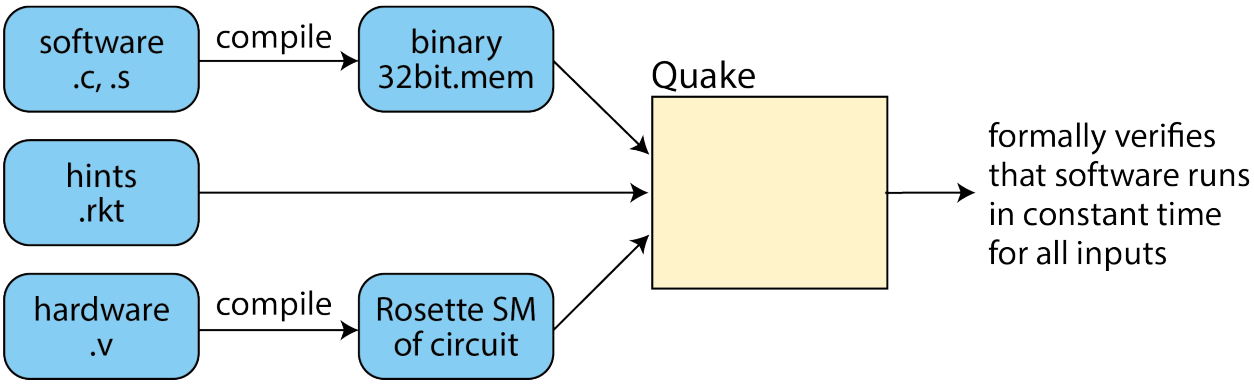

Figure 4.1: User inputs to Quake are the software, hardware, and performance hints. Software is compiled to a binary to be consumed by the verification framework. Hardware is compiled to a Rosette state machine of the circuit.

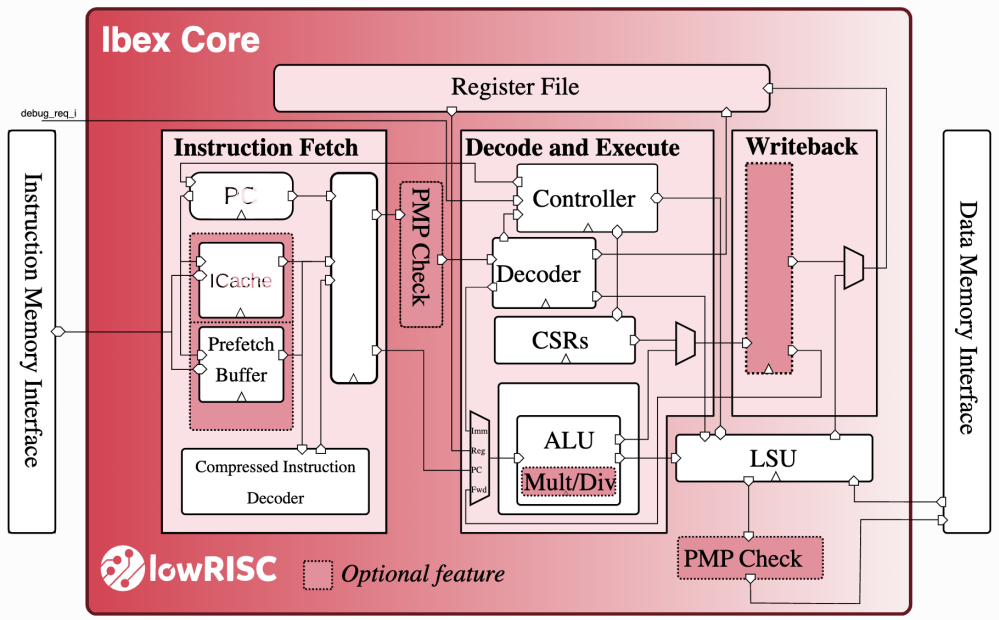

Figure 4.2: Ibex pipeline [13], which shows separate data and instruction memory interfaces

solver-aided programming in the verification framework.

4.1.1 SoC with Ibex Processor

Since different processors have different instruction and data interfaces that may affect how the ROM model is implemented, we select a specific processor to use while implementing Quake.

We used the Ibex, which is a small 32-bit RISC-V CPU core. It has a two-stage pipeline with an instruction fetch stage and an instruction decode and execute stage. It has an instruction-side memory interface used to fetch instructions and a data-side memory interface used to access data, as seen in figure 4.2. We chose the Ibex processor because we thought the separate instruction and data memory interfaces would be easier to reason about when implementing the ROM model.

The example hardware implementation used in this work is a SoC with the Ibex as its processor, seen in figure 4.3. The SoC also includes a RAM, FRAM, and GPIO, which interface with the Ibex processor. Since the ROM model in the Quake is provides the

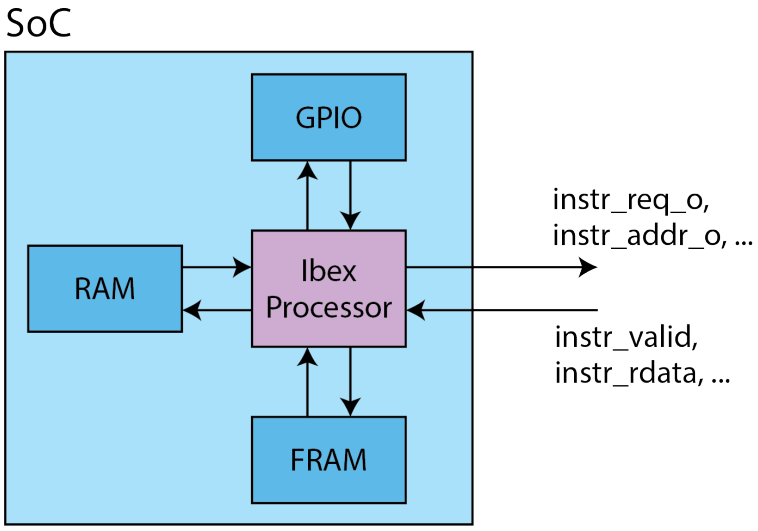

Figure 4.3: SoC with Ibex as a processor includes FRAM, RAM, and GPIO in its circuit, but does not include a ROM. Instead, it exposes the signals in listing 4.1

```
module soc #(
  ...
) (
    ...
    input [31:0] boot_addr_i,

    input instr_valid,
    input [31:0] instr_rdata,
    output instr_req_o,
    output [31:0] instr_addr_o,

    input rom_valid,
    input [31:0] rom_rdata,
    output mem_valid,
    output [31:0] mem_addr,
);
```

Listing 4.1: The SoC exposes the boot address, instruction-side memory interface and data-side memory interface signals at the top level of our circuit to allow the Rosette state machine to read and update these circuit state elements.

instructions, the example SoC does not include a ROM instead exposes the relevant signals from the instruction-side and data-side memory interfaces shown in listing 4.1

Finally, to take advantage of Rosette's support for solver-aided programming to do symbolic execution with circuits, developers compile the Verilog circuit to a Rosette state machine using the Verilog-to-Rosette toolchain from rtlv [11].

4.2 Software

Users supply the software they want to verify is constant-time to Quake. Since Quake verifies code is constant-time regardless of the memory address where the code is located, the supplied software needs to be compiled to be position-independent. It should execute correctly regardless of its absolute address.

The software examples in this work are written in C and assembly and are compiled to a binary using two compiler flags to ensure code is position-independent: `-fpic` and `-mcmodel=medany`.

4.2.1 Bootloader for Position-Independent Code

To handle code that can be located at arbitrary addresses, the bootloader for the software, shown in Listing 4.2, ensures that the stack pointer is initialized correctly. Quake uses concrete addresses for the RAM. Since the stack pointer points to an address in RAM, it should be concrete. Since the ROM is at an unknown address, the offset between the ROM and the RAM is unknown. However, RISC-V does not support text-relative data and enforces a fixed offset between the RAM and text.

To get the stack pointer, the bootloader loads in the `_stack_top` defined by the linker script, as seen in Listing 4.2. Since the compiler and linker assume a fixed offset between the ROM and RAM, `_stack_top` is loaded in with an `auipc` instruction that assumes the text begins at address 0. Since our code begins at a symbolic base address, this gives a symbolic

```
reset_handler:
    la sp, _stack_top
    auipc a0,0
    addi a0,a0,-144
    sub sp,sp,a0
    ...
```

Listing 4.2: Bootloader code for correctly initializing the stack pointer

```
void main() {
  uint32_t pc = getPc(); // reads the current instruction address
  if (pc == 1193400) {
      helper();
  } else {
      helper2();
  }
}
```

Listing 4.3: Example software that would use the hints function in 4.4 for verification

stack pointer.

To correct this, the bootloader first gets the base address by using the `auipc` instruction to get the program counter, then subtracting the current concrete offset. It then subtracts the symbolic base address from the initial stack pointer to get the concrete offset of the stack pointer from the start of the ROM, which is the correct stack pointer.

4.3 Hints

Users supply hints that specify circuit transformations that help with verification. They define and pass in a function that takes the state as a parameter and returns a list of hints to apply to the circuit state. The performance hint interface in Quake specifies the types of hints allowed.

Consider the example software in listing 4.3 which branches depending on the instruction address being executed. A developer may construct the `hints` function in listing 4.4 to support verification. The example `hints` function reads the program counter from the circuit

```
(define (hints state)
  ...
  (define pc (get-field state 'soc.cpu.u_ibex_core.if_stage_i.pc_id_o
     ))
  (match pc
    [(expression (== ite) c _ _)
     (list
      (hint:case-split (list c (! c)))
      (hint:concretize-with-function my_function))]
    [_ #f]))

(verify-timing state ... #:hints hints)
```

Listing 4.4: Example hint that applies the `case-split` and `concretize-with-function` hints when there are branching instructions

state. If the program counter is an if-then-else expression, meaning the execution branches, it applies two hints. The first hint is a `case-split` hint which accepts the list of cases as a parameter. For the branching instruction, the two cases are the if condition being true and the if condition being false. The second hint is a `concretize-with-function` hint which accepts a function that applies circuit transformations as a parameter. If the program counter is not an if-then-else expression, the `hints` function returns to indicate that no hints are needed in this cycle of execution. The implementation of these hints is described in section 5.2.

Chapter 5

Implementation for the Ibex Processor

This chapter describes how we implement Quake to verify cryptographic software is constant-time on a specific hardware implementation using a ROM model that gets instruction data from symbolic instruction addresses. Since different processors have different instruction and data interfaces, we implement Quake for the Ibex processor.

Quake works by applying a series of state transformations on each hardware cycle. It first performs any state transformations defined by the user-provided hints. Then, it uses the ROM model to update the state with the correct instruction data from the symbolic instruction address. And finally, it updates the state using ROM model-aware circuit state transformations to simplify symbolic expressions in the state.

This work builds off Chroniton [14], prior work that verifies timing properties of software on a specific hardware implementation. However, Chroniton only verifies end-to-end programs located at instruction address 0. We extend Chroniton to handle symbolic instruction addresses to verify that cryptographic library functions located at any instruction address run in a constant number of cycles.

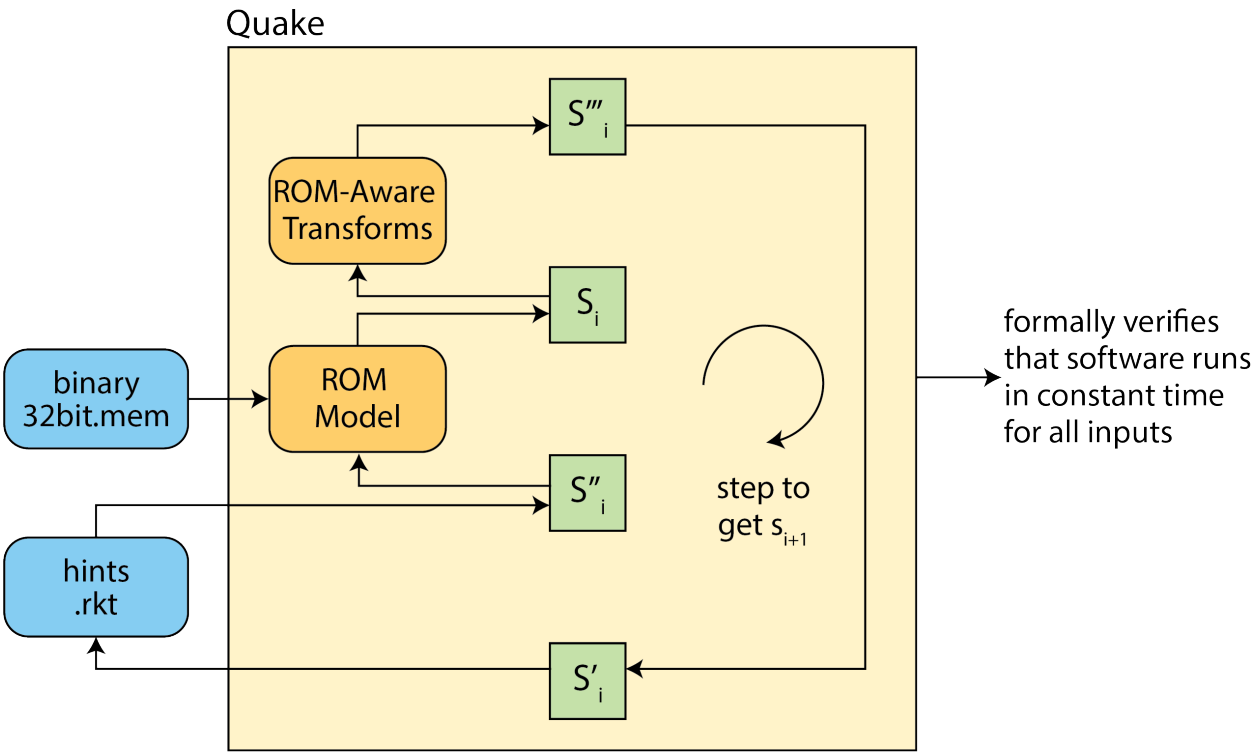

Figure 5.1: Diagram showing how Quake is implemented. s_i is the circuit state given by the step function that simulates one clock cycle. The s_i', s_i'', and s_i''' are circuit states that result from applying some circuit state transformation.

5.1 Initialization

On initialization, the software binary is loaded into the verification infrastructure. The binary is read and parsed into a Rosette vector of 32-bit bitvectors corresponding to assembly instructions. This vector is saved as `prog` in the verification infrastructure so that it can later be indexed to get instructions. The Rosette circuit `boot_addr_i` is then initialized with a symbolic boot address, which is also stored in the verification infrastructure as the `base_addr`.

Since our infrastructure has a symbolic ROM address, but concrete RAM and FRAM addresses, we enforce a fixed-size ROM to ensure that the location of ROM does not overflow into the concrete address spaces defined for RAM or FRAM by the SoC. FRAM is located at addresses with top byte 0x10, RAM is located at addresses with top byte 0x20, and we choose to locate ROM at memory addresses with top byte 0x00. As a result, the ROM has a fixed size of 0x01000000 or 2^{24} bytes.

We initialize the base address to be a symbolic 32-bit bitvector with top 9 bits 0 and bottom 8 bits 0. The bottom 8 bits are 0 since the Ibex requires that the boot address be aligned to 256 bytes [13]. The top 8 bits are 0 since we chose to locate ROM at memory addresses with top byte 0x00. If the 9th bit were symbolic, then the largest possible base address would be 0x00FFFF00, leaving only 256 bytes before an instruction is at address 0x01000000 and the program overflows from our 2^{24}-byte fixed-size ROM. To allow for a maximum $2^{23} + 256$ bytes of code, we constrain our base address to have top 9 bits 0.

5.2 Hints

Quake has a performance hint interface that allows for developers to provide hints on how to transform the circuit state during symbolic execution. Since developer-provided hints are untrusted, Quake is responsible for ensuring that incorrect hints do not cause the framework

```
def case-split(state, cases):
  pruned-cases = cases.filter(lambda c: state is satisfiable with
     constraint c)
  union-cases = union of pruned-cases
  if (state with constraint (complement of union-cases) is
     satisfiable):
    error: failed to prove exhaustiveness
  return cases.map(lambda c: (state, c))
```

Listing 5.1: pseudocode for the `case_split` hint

to incorrectly verify that a piece of software runs in constant time. The two main hints used by Quake are the `case-split` hint and the `concretize-with-function` hint.

5.2.1 case-split Hint

The `case-split` hint is useful for developers verifying software with branching instructions as seen in section 4.3. This hint (Listing 5.1) accepts as parameters the state and a list of cases that represent constraints on the state. It constructs `pruned-cases` by filtering out any case constraints that make the state unsatisfiable. This pruning is ensures that Quake does not branch unnecessarily. Then, it computes the constraint created by the union of these pruned cases. If the complement of this `union-cases` constraint applied to the state is satisfiable, the pruned cases do not cover all possible values in the state space. The `case-split` hint throws an error in this case, as hints should not reduce the space of possible circuit states to ensure correctness.

Otherwise, the function maps each case to a tuple of the state and the case constraint, which is the path condition for that branch. The step function which executes the next clock cycle and all state transformation functions accept an optional `predicate` or `pc` (for path condition). This parameter allows Quake to apply the path condition to all future solver queries on that branch. Finally, this list of states and their path conditions is used by Quake for the rest of the simulation. It executes each branch separately, applying the path condition to all queries for that branch. After all branches have completed, Quake verifies

that the number of hardware cycles executed is the same for all branches.

5.2.2 concretize-with-function Hint

The `concretize-with-function` hint allows for developers to pass in their own circuit state transformation function, which is particularly useful for any complex processor-specific transformations needed to execute with symbolic memory addresses.

The `concretize-with-function` hint accepts as a parameter a circuit state transformation function. Since hints are untrusted, Quake ensures that the function passed in does not reduce the space of possible circuit states. It does this by querying the SMT solver to check that the initial state and updated state with the function applied to it are provably equal. If the solver determines that the starting state and the state after the transformation are not equal, it throws and error and stops the simulation. Two states are provably equal if there is no possible assignment of concrete values that satisfies one of the states, but not the other state. For example, the symbolic expression `(ite (= 0 0) sym_1 sym_2)` is symbolically equivalent to the symbolic expression `sym_1`. The first expression can be simplified to `sym_1` and there are no concrete values that satisfy `sym_1`, but not `sym_2`. Similarly, the transformation we apply to instruction addresses `instr_addr_o` gives `concretize(instr_addr_o - base_addr) + base_addr` which is provably equal to `instr_addr_o`. The `concretize` function finds a single concrete value that its parameter evaluates to, so `concretize(instr_addr_o - base_addr) = instr_addr_o - base_addr` and the whole expression evaluates to `instr_addr_o`.

The requirement that the function passed in returns a state that is provably equal to the initial state is stronger than the requirement that hints do not reduce the space of possible circuit states. Consider an overapproximate hint replaces a symbolic expression with a fresh, unconstrained symbolic, which yields an expression satisfied by a superset of concrete values that satisfy the original expression. This is okay since we can always overapproximate a value as it can only increase the space of possible circuit states and therefore, increase variation in

timing behavior. However, writing a check that the updated state with the function applied is a superset of the initial state is more complicated and not strictly necessary to achieve for our verification goals. Quake instead imposes a stronger precondition on clients of the `concretize-with-function` hint.

5.3 ROM Model

Quake replaces the ROM in the circuit with a ROM model in the verification infrastructure. The ROM model handles getting data from symbolic memory addresses using its offset from the symbolic base address. It does this for both the instruction-side memory interface to read the next assembly instruction to execute and the data-side memory interface to read any global constants stored in the ROM.

5.3.1 Instruction-Side Memory Interface

For the instruction-side memory interface, with code shown in Listing 5.2, Quake gets the `instr_addr_o`, which is the instruction address, and `instr_req_o`, the request valid bit, from the circuit state. These are the output signals of the instruction-side memory interface seen in Listing 4.1

The offset is computed by subtracting the stored symbolic `base_addr` from the symbolic `isntr_addr_o` and calling the `concretize` utility function on the result. Quake uses the concrete offset to index into the `prog` bitvector to get the instruction data, which is used to update `instr_rdata` in the circuit.

The `instr_valid` element also depends on the instruction address, so it is also updated. It is set to true if `instr_req_o` bit is true and the top byte of the instruction address is 0x00, meaning the address is in the ROM's address space.

```
base_addr = new_symbolic()
program = []

def step(circuit_state):
  ...
  updated_circuit = read_rom_model_instruction(circuit_state)
  ...

def read_rom_model_instruction(circuit_state):
  offset = concretize(circuit_state.instr_addr_o - base_addr)
  instr_addr = base_addr + offset
  instr_rdata = prog[offset]
  instr_valid = circuit_state.instr_req_o && instr_addr[31:24] == 0x0
  return circuit_state.update(instr_rdata, instr_valid)
```

Listing 5.2: Pseudocode that shows how the ROM model is used to read concrete memory data from symbolic memory addresses for the instruction-side interface and update the circuit state

5.3.2 Data-Side Memory Interface

The data-side memory interface is very similar to the instruction-side memory interface, with the main difference that often `mem_valid` is false, in which case Quake does not examine the symbolic data-side memory address, `mem_addr`, and try to concretize its offset from the symbolic base address. If `mem_valid` is false, Quake updates the state with `rom_valid` set to false.

It is important to call the `concretize` function on `mem_valid` before checking if it is false as `if(mem_valid)` will be true if `mem_valid` is some symbolic expression, even if it evaluates to false. Skipping this step can make verification intractable if it leads to an attempt to read data from a symbolic offset.

If `mem_valid` is true, Quake determines the offset of the symbolic `mem_addr` from the symbolic base address. It determines if `rom_valid` is true if the top byte of `mem_addr` is 0x00. Since `mem_addr` is also used to index into RAM and FRAM, `rom_valid` is often false. In that case, `rom_rdata` will not be used, so Quake returns 0x00000000 rather than wasting time reading data from a possibly symbolic offset. Finally, `rom_valid` and `rom_rdata` are

```
base_addr = new_symbolic()
program = []

def step(circuit_state):
  ...
  updated_circuit = read_rom_model_data(circuit_state)
  ...

def read_rom_model_data(circuit_state):
  if concretize(circuit_state.mem_valid):
    offset = concretize(circuit_state.mem_addr - base_addr)
    mem_addr = base_addr + offset
    rom_valid = mem_addr[31:24] == 0x0
    rom_rdata = rom_valid ? prog[offset] : 0x00000000
    return circuit_state.update(rom_rdata, rom_valid)
  else:
    rom_valid = false
    return circuit_state.update(rom_valid)
```

Listing 5.3: Pseudocode that shows how the ROM model is used to read concrete memory data from symbolic memory addresses for the data-side interface and update the circuit state

updated in the circuit.

5.4 ROM-Aware Circuit State Transformations

Other circuit state elements besides the memory data may also depend on the instruction address. For example, the Ibex processor has an instruction prefetch buffer for performance that stores the instructions themselves with the instruction address they came from.

Since Quake uses symbolic instruction addresses and other circuit state elements may depend on the instruction address, the size of symbolic expressions in the state grows as operations are executed, making solver queries intractable. To avoid timing out during circuit simulation, Quake performs various circuit state transformations that simplify symbolic expressions.

These circuit state transformations are processor-dependent as they are specific to how a particular processor uses memory addresses. The following cases appear for the Ibex

processor and the heuristics for when to apply them were sufficient for making verification finish in a reasonable time.

Quake loops over all circuit state elements and attempts to simplify any symbolic expressions that depend on the base address. There are three heuristic approaches we take depending on the size of the bitvector:

1. If a circuit state element is a 32-bit bitvector, Quake attempts to simplify its symbolic expression in the same way it simplified the instruction address. It subtracts the base address from the field and queries the SMT solver with the `concretize` function. If this offset is concrete, the field is updated to be the sum of the base address and the offset.

2. If a circuit state element is a 31-bit bitvector, it may be a concrete offset from the 31-bit prefix of the base address. Quake subtracts the 31-bit prefix of the base address from the field and queries the SMT solver with the `concretize` function. If this offset is concrete, the field is updated to be the sum of the top 31 bits of the base address and the offset.

3. Finally, if either of the previous checks failed to evaluate the offset to single concrete value or the circuit state element is a different size, then Quake queries the SMT solver with the `concretize` function on the field directly and updates the field to be this concrete value if successful.

The Rosette state machine is updated with the simpler symbolic expressions for these circuit state elements. Performing this concretization at each step of the circuit helps reduce the size of solver queries in future cycles of the circuit.

Quake does the state transformations in this order to minimize calls to `concretize` as SMT solver queries can be slow. Empirically, the 32-bit circuit state elements are sometimes evaluated to a single concrete value (approach 3), but more often are a concrete offset from the base address (approach 1), so Quake attempts that first. And empirically, the 31-bit circuit

state elements are always a concrete offset from the prefix of the base address (approach 2) and never evaluated to a single concrete value (approach 3).

This ordering breaks when the base address can be evaluated to a single concrete value, but the stored base address is symbolic. This may happen when execution branches (see section 5.2.1) and Quake is executing with a path condition that is used for all `concretize` function calls. Since the base address can be evaluated to a single concrete value, a 32-bit circuit state element that can be evaluated to a concrete value also has an offset from the base address that can be evaluated to a concrete value. If Quake attempts approach 1 first, it finds that the offset is concrete and updates the circuit state element to be that offset plus the stored symbolic base address, which is not the most simple form of the expression. Instead, Quake checks if the base address can be evaluated to a single concrete value at the start of this function. If so, it only uses approach 3 to simplify circuit state elements.

Chapter 6

Performance Enhancements

This section discusses two attempts to improve performance, overapproximating and using an LRU cache. Overapproximating improved performance and is implemented in Quake, whereas the LRU cache did not improve performance.

6.1 Overapproximating

We found that as the simulator executed, some symbolic terms would grow rapidly in size with more computations. For Ed25519, some circuit state elements would depend on 10 symbolic values by cycle 2737 and by cycle 3331, some elements depended on 30 symbolic values. To reduce the size of solver queries, Quake applies the `overapproximate` utility function from rtlv [11] to any circuit state element whose symbolic expression depended on more than one symbolic value. This function gets a fresh, unconstrained symbolic variable, which is used to update the circuit state. Since removing restrictions on symbolic variables gives a superset of possible circuit states, doing this optimization does not lead to missed variations in timing behavior. Additionally, Quake does not overapproximate circuit state elements that depend on the base address as those expressions could be concretized by the ROM model.

A small experiment of simulating an implementation of Ed25519 on Ibex for 5000 cycles

Base Address	Without Overapproximate	With Overapproximate
concrete	417.6	47.9
symbolic	13399.8	714.5

Table 6.1: Verification time (seconds) for 5000 cycles of Ed25519 with and without overapproximate. For concrete base addresses, overapproximate gives an 8.7x speed improvement and for symbolic base addresses, overapproximate gives a 2.0x speed improvement.

showed an 8.7x speed improvement for concrete base addresses and a 2.0x speed improvement for symbolic base addresses, as shown in table 6.1. It is reasonable that the speed improvement is greater for concrete base addresses, as the verification time for symbolic base addresses is more likely dominated by queries to the SMT solver.

6.2 LRU Cache

For symbolic base addresses, we noticed that the majority of verification time (over 60%) was spent in the `concretize` function which queries the SMT solver. Quake calls this function when applying ROM model-aware circuit state transformations and in applying the ROM model when it tries to concretize the offset between the base address and memory address.

We attempt to minimize time spent in the `concretize` function by reducing the number of times it's called. However, we would not want to remove a `concretize` function call that successfully evaluates the symbolic expression to a concrete value as the performance cost of having very large solver queries is greater than one `concretize` call. Examining our simulation of Ed25519, there are almost no unsuccessful `concretize` calls. All attempts to concretize the offset between the base address and instruction or memory address are successful. And some careful logging indicates that all circuit state elements that depend on the base address can be concretized via one of the three approaches implemented. The only case when a `concretize` call fails is when the symbolic expression evaluates to a concrete value, but the element is a 32-bit bitvector. In this case, Quake attempts to call `concretize` on its offset from the base address first, which fails, before calling `concretize` on the expression

itself. However, switching the order of this check would not help as the case that a 32-bit bitvector is simplified by calling `concretize` on its offset is more frequent than the case that its symbolic expression evaluates to a concrete value.

One way we may reduce the time spent in the `concretize` function is by caching the results to the `concretize` call and checking the cache before calling `concretize`. We thought caching the results of the `concretize` calls for arbitrary circuit state elements would be less likely to improve performance as there are many circuit state elements and their values change as the program executes. On the other hand, we thought caching the `concretize` results for memory addresses would be more likely to improve performance as the number of addresses accessed is bounded and the Ed25519 example has loops that reuse recently accessed addresses. So we experimented with an LRU address cache that maps symbolic memory addresses to their concrete offset from the base address. We chose a cache bound of 1024 based on the size of the loops in the Ed25519 code.

Our cache had about a 10% hit rate when simulating Ed25519. Though we did not run a full experiment comparing verification time for Ed25519 with and without the LRU cache, smaller experiments with a few thousand cycles showed that the version with the LRU cache actually ran about 7% slower than the original version. For this example, the cache did not reduce the `concretize` function calls enough to justify the extra time required to maintain it. It may be useful for other functions and is easily toggled in the verification infrastructure.

Chapter 7

Evaluation

We evaluate Quake with the following questions:

1. Is Quake able to detect timing variations that depend on the instruction address code is located?

2. How does removing the ROM from the circuit and replacing it with a ROM model in the verification infrastructure affect verification time? How does making the instruction address symbolic affect verification time?

3. How easy is it to remove ROM for a new system on a chip?

All experiments are done on an Intel i7-5930K.

7.1 Detecting Timing Variations with Base Address

Since our goal is to verify that cryptographic library functions are constant-time at a cycle-level regardless of the instruction address the code is located, we want to check that we can detect timing variations that depend on where the code is located. To evaluate this, we construct an example that branches depending on the value of the current instruction address, shown in Listing 7.1 and 7.2.

```
void main() {
  uint32_t pc = getPc();
  if (pc == 1193400) {
      helper();
  } else {
      helper2();
  }
}
```

Listing 7.1: main function for code with timing behavior that varies with the base address

```
.globl getPc
getPc:
  auipc a0, 0
  ret

.globl dummy
dummy:
  add a3, a3, a2
  add a4, a4, a1
.rept 500000
  nop
.endr
  ret

.globl helper2
helper2:
  add a5, a4, a3
  ret

.globl helper
helper:
  add a4, a3, a2
  ret
```

Listing 7.2: assembly helpers for code with timing behavior that varies with the base address

The example we construct uses a small assembly helper, `getPc` in Listing 7.2 to get the current program counter. If the program counter is a specific concrete value, then it takes one branch. Otherwise, it takes a different branch. Each branch calls a small helper that performs one add instruction. Since the Ibex takes two extra cycles to execute when a branch is taken, this example has timing behavior that depends on the location of the code in memory.

Futhermore, to model a library function that cannot use absolute addresses, we construct an example that might use absolute addresses and ensure that it is correctly compiled to be position-independent. Code can have relative jumps, which jump to the address `pc + offset` where the offset is given by the 20-bit immediate in the instruction. Since these jumps are relative to the current instruction address, relative jumps are always position-independent. If the destination address is too far such that the offset does not fit into the immediate address, the code needs to use an indirect jump instead, which loads the destination address from a register. If the absolute address of the destination is loaded into the register, then the code is not position-independent.

Thus, to ensure that our example is compiled to position-independent code correctly, we force it to use an indirect jump. Since the offset for a relative jump must fit in 20 bits and each instruction is 4 bytes long, the maximum relative jump is $2^{20}/4 = 262144$ lines away. To force the helper functions to be at least 262144 lines away from `main`, we add a dummy function to our assembly file before the helpers that has 500000 `nop` instructions.

Quake successfully detects the timing attack due to the branching on the instruction pointer (7.3), using the `case-split` and `concretize-with-function` hints, similar to the example Listing 4.4. The `case-split` allows Quake to execute two simulations, one for each branch. Then the function we pass into the `concretize-with-function` does the ROM model-aware circuit state transformations described in 5.4. These transformations use the path condition to simplify the circuit state as much as possible, which allows the verification to run in only 35 seconds.

```
verify-timing: mismatch in number of cycles on different branches (80
    vs 78)
  context...:
   /chroniton/chroniton/main.rkt:21:0: verify-timing
   body of "/chroniton/examples/ibex-branch-addr.rkt"
```

Listing 7.3: Output when attempting to verify our example that branches depending on the value of the current instruction address

7.2 Verification Time

We compare the verification time for software using a ROM in the circuit with the verification time for software using our ROM model. Since the ROM model allows us to use symbolic base addresses, we also compare the verification time for the software using a ROM model with a concrete base address with software using a ROM model with symbolic base address.

Since we want to see how practical this approach is for real cryptographic algorithms, we verify a C implementation of Ed25519 [15], an elliptic curve signing algorithm used in practice for public-key cryptography. We compare the verification times for Ed25519 in table 7.1. Using the ROM model with concrete addresses made the verification time 4.5x slower than having the ROM in the circuit. This increase in verification time makes sense: having the ROM in the circuit allows for the step function in the Rosette circuit to determine the correct instruction from the address in the circuit, whereas the ROM model requires some computation at every cycle to determine the appropriate offset, read the data from that offset and update the state with the correct values.

When looking at verification time using a ROM model with symbolic addresses compared to using a ROM model with concrete addresses, the verification time is 6.6x slower for the Ed25519 example. This increase in verification time is reasonable as using a symbolic base address requires Quake to query the SMT solver to simplify the symbolic expressions for addresses as well as other circuit state elements.

Since the purpose of using the ROM model is to allow for symbolic addresses, we compare

ROM	Addresses	Verification Time	
In Circuit	Concrete	3.20	
ROM Model	Concrete	14.34	(4.5x)
ROM Model	Symbolic	94.95	(29.6x)

Table 7.1: Verification time (hours) for Ed25519 with the ROM in the circuit, a ROM model with a concrete base address, and a ROM model with symbolic base address

the verification time using the ROM model with symbolic addresses to the verification time using the ROM in the circuit. Using the ROM model with symbolic addresses was 29.6x slower than using the ROM in the circuit for Ed25519. Since the goal of this work is to verify the timing behavior of a piece of software and there is not a particularly strong need to do it in the fastest time possible, this slowdown is acceptable as it still allows the verification to run in a reasonable amount of time (4.0 days).

7.3 Implementation Effort

We want to understand how easy it is to remove the ROM and implement a ROM model for a system on a chip using a new processor. We proxy this proof effort by comparing the lines of processor-specific code needed in our verification infrastructure, shown in table 7.2.

As we can see, using a ROM model does require more processor-specific logic in the verification infrastructure as it requires ensuring that the memory interface between the Rosette circuit and the ROM model is handled correctly. The implementation also requires calling `concretize` in the correct places to make sure solver queries do not become too large with symbolic memory addresses. Since we only experimented with one processor, it is not clear how this implementation effort would change with more complicated processors or those that have a different memory interface. For example, the PicoRV32 is a simple processor that has one memory interface shared between data and instructions, so writing a ROM model for the PicoRV32 may require logic to distinguish instruction and memory addresses.

Processor	ROM in Circuit	ROM Model
Ibex	31	181

Table 7.2: LOC that are processor-specific in Quake

Chapter 8

Discussion and Future Work

This work makes some progress towards cycle-level verification of constant-time cryptography by verifying software is constant-time regardless of the instruction address of the code. There are still some limitations to the approach if we want to verify arbitrary library functions running in any context.

8.1 Hardware Limitations

8.1.1 Other Processor State

Since library code can be run at an arbitrary point in the execution of a program, the processor's state at the start of execution of a library function can be arbitrary. For example, there may be existing register values in the processor, data on the stack, or other instructions in the processor's pipeline. Currently, Quake only considers starting from a reset processor state, which would not catch any timing variability due to the existing processor state.

8.1.2 Cache

Previous timing side channel attacks, such as Meltdown and Spectre [cite] have taken advantage of cache state and the timing differences between loading cached and uncached values

to bypass protection checks. Since caches are implemented to improve timing, timinig verification that handles cache state would be particularly useful. Since we only experimented with using a ROM model with the Ibex processor, which does not use a cache, we were not able to observe the effects of caching on timing behavior.

8.1.3 RAM Location

While Quake handles different locations for code, it does not vary the location of RAM in the address space. Modern operating systems do address space layout randomization (ASLR) to randomly arrange various data areas including libraries, executables and the RAM. To more closely simulate software run in a randomized address space layout, we would want to also account for variations in the location of the RAM.

Furthermore, our current implementation of the ROM model assumes that the base address of the ROM is prefixed by 0x00. We do this to avoid colliding with the addresses spaces of FRAM and RAM, which are located at concrete addresses starting at 0x10000000 and 0x20000000, respectively. Future work could not only handle different starting addresses for RAM and FRAM within their bounds, but also model address space layout randomization.

8.2 Software Limitations

8.2.1 Cannot Use Data Section

One software limitation of our current implementation is that it does not allow for software that uses the .data section for global variables. It only allows for software to use the .rodata section for static constants and Ed25519 does read from this section.

We have this limitation because our code is compiled for RISC-V which does not support text-relative data and enforces a fixed offset between the RAM and code regions. Since our implementation varies the address of code, but does not move the RAM, we cannot

determine the address of the RAM relative to the instruction address. Since the GOT is located in RAM, this also means that we cannot verify software that uses the GOT.

Chapter 9

Related Work

9.1 Software Verification with a Leakage Model

The standard approach to verifying software uses leakage models. Almeida et. al. define constant-time to mean that any two executions of a program with the same public inputs and outputs leak exactly the same observations according to their leakage model [16]. They use a leakage model that includes branches and memory access locations [5]. Likewise, Bond et. al. introduce a tool Vale that uses the execution trace, which records branches, memory-access locations, and inputs to variable-latency instructions—their leakage model—to verify programs free of timing side-channels. Code is deemed leakage-free if two executions that have the same initial public state and initial execution trace have the same final public state and final execution traces [6]. HACL* is a verified cryptographic library written in F* that depends on the type system to only allow verified primitive operations on secrets, which are assumed to be constant-time.

While leakage models are a useful abstraction, there are many different leakage models that assume different threat models. Cryptographic software libraries such as OpenSSL often provide different implementations for different leakage models [2]. Ammanaghatta et. al. discuss the challenges of having many leakage models and implementations: it's harder to

ensure a specific implementation is verified according to the correct leakage model, especially as some are easier to verify than others. They cite the Lucky13 timing attack, which took advantage of routines that were provably secure under the baseline leakage model, which only leaks the program counter and memory operation accesses, but not secure under the time-variable model, which leaks time-variable arithmetic operations [17]. Furthermore, leakage models do not account for differing hardwares. For example, HACL* uses integer multiplication even though some ARM and i386 processors do not have constant-time integer multiplication, [7].

9.2 Processor Verification with a Leakage Model

There are other efforts to bridge the gap between leakage models and hardware, such as the work by Wang et. al. that focuses more on the hardware side by verifying that processors satisfy leakage contracts [9]. They do this by modeling the leakage contract and the attacker as monitoring circuits that produce traces of their leakage. The contract is satisfied for a given attacker if two architectural states that lead to executions distinguishable by the attacker circuit also lead to executions distinguishable by the contract circuit. Their approach is effective at characterizing the security guarantees provided by three simple RISC-V processors.

9.3 RISC-V Verification of Processors

Riscv-formal is a tool for verifying that a processor implements the RISC-V ISA. It focuses on verifying correctness with respect to an ISA, but does not consider timing or make any security claims.

9.4 Chroniton

Chroniton is a tool for verifying constant-time behavior of cryptographic software by simulating software on a particular hardware implementation at the RTL level. Chroniton only verifies a program loaded at ROM address 0 and executed from end-to-end. It makes no claims about library functions or code fragments executed in an arbitrary context or from an arbitrary instruction address. This work extends Chroniton to support a ROM model that allows for execution of code located at symbolic memory addresses.

Chapter 10

Conclusion

This thesis presents Quake, a verification framework for verifying that cryptographic software is constant-time regardless of the location of code. It uses a ROM model that determines the instruction data from a symbolic instruction address. This thesis also implements Quake for the Ibex processor. And finally, this thesis uses Quake to verify Ed25519 is constant time regardless of the address of the code and examines the limitations and challenges associated with verifying constant-time cryptography with this approach.

Bibliography

[1] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, *Hacl*: A verified modern cryptographic library*, Cryptology ePrint Archive, Paper 2017/536, https://eprint.iacr.org/2017/536, 2017. URL: https://eprint.iacr.org/2017/536.

[2] B. A. Shivakumar, G. Barthe, B. Grégoire, V. Laporte, and S. Priya, *Enforcing fine-grained constant-time policies*, Cryptology ePrint Archive, Paper 2022/630, https://eprint.iacr.org/2022/630, 2022. URL: https://eprint.iacr.org/2022/630.

[3] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C.: USENIX Association, Aug. 2003. URL: https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical.

[4] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Proceedings of the 16th European Conference on Research in Computer Security*, ser. ESORICS'11, Leuven, Belgium: Springer-Verlag, 2011, pp. 355–371, ISBN: 9783642238215.

[5] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, *Verifiable side-channel security of cryptographic implementations: Constant-time mee-cbc*, Cryptology ePrint Archive, Paper 2015/1241, https://eprint.iacr.org/2015/1241, 2015. URL: https://eprint.iacr.org/2015/1241.

[6] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying High-Performance cryptographic assembly code," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 917–934, ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond.

[7] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1789–1806, ISBN: 9781450349468. DOI: 10.1145/3133956.3134043. URL: https://doi.org/10.1145/3133956.3134043.

[8] T. Pornin. "The problem." (2018), URL: https://www.bearssl.org/ctmul.html (visited on 12/14/2023).

[9] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, *Specification and verification of side-channel security for open-source processors via leakage contracts*, 2023. arXiv: 2305.06979 [cs.CR].

[10] Microsoft, *Z3*, https://github.com/Z3Prover/z3, 2024.

[11] N. Moroze, A. Athalye, M. F. Kaashoek, and N. Zeldovich, "Rtlv : Push-button verification of software on hardware," 2021. URL: https://api.semanticscholar.org/CorpusID:235468913.

[12] C. X. Wolf, *Yosys*, https://github.com/YosysHQ/yosys, 2023.

[13] lowRISC, *Ibex*, https://github.com/lowRISC/ibex, 2024.

[14] anishathalye, *Chroniton*, https://github.com/anishathalye/chroniton, 2023.

[15] orlp, *Ed25519*, https://github.com/orlp/ed25519, 2022.

[16] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16, Austin, TX, USA: USENIX Association, 2016, pp. 53–70, ISBN: 9781931971324.

[17] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: Breaking the tls and dtls record protocols," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 526–540. DOI: 10.1109/SP.2013.42.