Providing High-Performance Execution with a Sequential Contract for Cryptographic Programs

Ali Hajiabadi
National University of Singapore

Trevor E. Carlson
National University of Singapore

ABSTRACT

Constant-time programming is a widely deployed approach to harden cryptographic programs against side channel attacks. However, modern processors violate the underlying assumptions of constant-time policies by speculatively executing unintended paths of the program.

In this work, we propose CASSANDRA, a novel hardware-software mechanism to protect constant-time cryptographic code against speculative control flow based attacks. CASSANDRA explores the radical design point of disabling the branch predictor and recording-and-replaying sequential control flow of the program. Two key insights that enable our design are that (1) the sequential control flow of a constant-time program is constant over different runs, and (2) cryptographic programs are highly looped and their control flow patterns repeat in a highly compressible way. These insights allow us to perform an offline branch analysis that significantly compresses control flow traces. We add a small component to a typical processor design, the *Branch Trace Unit*, to store compressed traces and determine fetch redirections according to the sequential model of the program. Moreover, we provide a formal security analysis and prove that our methodology adheres to a strong security contract by design. Despite providing a higher security guarantee, CASSANDRA counterintuitively improves performance by 1.77% by eliminating branch misprediction penalties.

1 INTRODUCTION

Protecting cryptographic programs has always been a major concern since they are the primary secret processing programs. While cryptographic schemes provide strong security levels to prevent secret extraction through cryptanalysis, their implementations are still vulnerable to various side channel attacks. *Constant-time* programming is a widely deployed approach to protect cryptographic programs. Constant-time principles mandate the absence of secret dependent control flow and data flow. In other words, the dynamic control flow and data flow of the program must be independent of the confidential inputs of the program [2].

Unfortunately, modern processors violate constant-time principles which assume instructions are executed sequentially (i.e., sequential execution model). Speculative execution attacks have demonstrated the ability to leak secrets from verified constant-time programs by transiently declassifying and leaking confidential states [48, 58]. Existing defenses to protect constant-time programs, both on the hardware level [12, 14, 43] and the software level [11, 34, 48, 49, 55, 61], deploy a restrictive approach to prevent or limit speculative execution of instructions, diminishing the benefits of speculative, out-of-order processors.

While existing CPUs can efficiently mitigate data flow speculation for cryptographic programs [34], addressing the speculation that arises from the control flow of the program is still a major issue. We investigate a new, radical design point to address this problem, namely recording-and-replaying; this mechanism records the sequential control flow trace of the program and redirects the fetch based on these traces, instead of prediction. This design ensures that fetch is always redirected according to the sequential execution model of the program, as assumed by constant-time policies, eradicating the dangers of control flow misspeculations. However, this idea has two major challenges:

Challenge 1: Dynamic control flow traces change based on the program input; pre-computing control flow traces for all possible inputs in general-purpose applications is challenging, if not infeasible.

Challenge 2: Control flow traces can be huge and communicating and loading these traces in the processor would incur high overheads. In the worst case, it can show similar slowdown as a processor without a branch predictor which stalls fetch until the branch condition is resolved.

However, we discuss two key insights from constant-time cryptographic programs that overcome these challenges:

Insight 1: Sequential control flow of constant-time programs are constant w.r.t. confidential inputs. In addition, public parameters of cryptographic programs are specified by standards or determined by the algorithm (e.g., the key length, number of encryption rounds, etc.). Hence, reusing just a single control flow trace over different runs of a program can be sufficient. However, control flow traces can still reach up to millions of decisions per static branch. As

mentioned in **Challenge 2**, storing and communicating a huge number of decisions per branch is not efficient, and a solution is needed.

Insight 2: Most operations in cryptographic programs are wrapped in loops and they repeat the same operations over time. Detecting the repeating patterns of branch decisions would help to allow the storage of smaller, compressed patterns, and once loaded, the processor can replay the same pattern in the future.

In this paper, we propose CASSANDRA, a defense for cryptographic programs against control flow based Spectre attacks. To the best of our knowledge, CASSANDRA is the first defense that exploits the key characteristics of cryptographic applications, and counterintuitively, *improves* performance. The main artifacts of CASSANDRA are twofold:

(1) Branch analysis (§4). We perform an extensive branch analysis of cryptographic programs and devise a trace compression technique that significantly compresses branch traces. Our approach is similar to DNA sequencing techniques that detect frequent and unknown patterns of nucleotides in large DNA sequences [33]. The average size of our new traces is 21 entries and less than 10 for most applications in BearSSL library [3].

(2) Microarchitecture (§5). We propose a new design that (1) communicates compressed branch traces to the processor, and (2) uses branch traces for fetch redirections and avoids accessing and updating the branch predictor. We add a small, new component to the frontend, called the *Branch Trace Unit* (*BTU*), that efficiently stores and decompresses dynamic branch information.

In addition, we provide a formal security analysis to express CASSANDRA's security guarantees (§6). We build our formalization on top of hardware-software contracts [16] and prove security for a contract that only leaks constant-time observations in a sequential execution model. Our main intuition is to design new hardware with a strong contract in mind and then optimize the design for high efficiency. We demonstrate that the combination of our key insights from cryptographic programs and trace compression enables efficient implementation of CASSANDRA's semantics.

The main contributions of CASSANDRA are as follows:

- Mitigating control flow based misspeculation in cryptographic code, while improving performance by 1.77% compared to an unprotected out-of-order processor;
- A branch trace compression technique, inspired by DNA sequencing methods, that significantly compresses traces;
- An efficient design of CASSANDRA that communicates branch traces with the hardware and enforces branch directions of a sequential contract;
- Providing a formal security analysis and proving CASSANDRA's security for a constant-time sequential contract.

```
1 uint8 decrypt(uint8 m, uint8 *skey)
2 {
3     uint8 state = m; //m and state are secret
4     for (int i = 0; i < num_rounds; i++)
5         state = decrypt_ct(state, skey[i]);
6     uint8 d = declassify(state); //d is public
7     return leak(d);
8 }
```

Listing 1: Constant-time decryption of `m`. Misspeculation and skipping the `for` loop can directly leak the secret `m`.

2 BACKGROUND

2.1 Constant-Time Programming

Modern cryptographic programs deploy constant-time policies to harden programs against traditional side channels that exploit secret dependent behaviors of the program. Constant-time principles satisfy confidential input indistinguishability to remove timing, cache, and memory side channels [2].

Executing a given program p with input x generates the attacker-visible execution trace θ:

$$\theta(p(x)) = [O_0 \cdot O_1 \cdot \ldots \cdot O_n]$$

where O_i represents the adversary's observation. Constant-time principles assume that the adversary can observe the program counter, memory access patterns, and operands of variable-time instructions [2].

Definition 1 (Constant-time programs). Given a program p with confidential input x and public input y, the observations of the execution are represented as $\theta(p(x, y)) = [O_0 \cdot \ldots \cdot O_n]$. Constant-time principles require that:

$$\forall y \in Data_{pub}, \forall x, x' \in Data_{conf}: \theta(p(x, y)) \simeq \theta(p(x', y))$$

where $\simeq$ denotes observation indistinguishability and $Data_{pub}$ and $Data_{conf}$ refer to the input space of public and confidential values, respectively [2].

Constant-time policies provide security for a *sequential execution* model, i.e., all instructions are executed in a sequential order specified by the architectural states of the program. However, Spectre-type attacks have demonstrated the ability to leak secrets from constant-time programs in modern processors when using a *speculative execution* model [48, 49, 58]. For example, Listing 1 shows a constant-time decryption of confidential input `m`. Sequential execution of the code dictates that after finishing all rounds of the decryption the secret state is declassified (line 6) and can legally leak (i.e., it is considered public after decryption, line 7). However, in a speculative execution model, the `for` loop can be skipped due to misspeculation and directly leak the confidential input `m` before executing all decryption rounds, and violate constant-time policies of the program.

2.2 Speculation Primitives

Speculative execution of programs can be triggered through different sources (referred to as speculation primitives) in modern out-of-order (OoO) processors. Speculation primitives can be categorized into control flow and data flow primitives [7, 10].

Control flow speculation. The Branch Prediction Unit (BPU) in modern processors predicts the next PC upon control flow instructions and fetches instructions speculatively from the predicted path. Control flow prediction allows the processor to avoid frontend stalls for cases where resolving control flow conditions depends on long latency operations. Prior attacks have demonstrated leaks via three main components in the BPU:

PHT The Pattern History Table (PHT) predicts *conditional direct branches* (e.g., `cmp [reg], 0; je L`) with two possible outcomes of Taken and Not-Taken. Spectre-v1 [24] is an example of exploiting the PHT primitive.

BTB The Branch Target Buffer (BTB) predicts *indirect branches* (e.g., `jmp [reg]`) to determine the target address of next instruction. Spectre-v2 [24] is an example of exploiting the BTB primitive.

RSB The Return Stack Buffer (RSB) predicts the target address of return instructions. While returns are also considered to be indirect branches, most processors use RSB to determine return addresses. Spectre-RSB [25] is an example of exploiting the RSB primitive.

Throughout this paper, we refer to all control flow instructions (direct, indirect, and return) as *branches*.

Data flow speculation. Modern processors deploy mechanisms for speculative execution of loads:

STL Store-to-load forwarding (STL) allows a load to forward data from a prior same-address store before all prior stores are resolved, without sending a request to the memory. Spectre-v4 [18] is an example of exploiting the STL primitive.

PSF Predictive store forwarding (PSF) allows a younger load to forward data from an unresolved store before the load and store addresses are resolved. Spectre-PSF [9] is an example of exploiting this primitive.

Mitigating control flow speculation primitives poses higher overheads compared to data flow primitives. Experiments from Mosier et al. [34] show that naively addressing data flow speculation by setting the SSBD control bit [20] in existing Intel CPUs incurs negligible performance overhead (less than 1% when no other compiler mitigation like SLH [8] or retpoline [51] is enabled for control flow speculation).

2.3 Evolution of Spectre Defenses

Early defenses for speculative execution attacks focused on data caches as the transmission channel of Spectre-v1 [23, 36, 41, 42, 57]. More comprehensive solutions, like STT [59] and NDA [56], proposed secure speculation mechanisms to prevent the leaks from speculatively loaded data via a more comprehensive list of transmission channels (i.e., *sandboxing policy*). These solutions implement dynamic taint tracking to restrict the execution or data propagation of the instructions that are tainted by speculatively loaded data. While this approach protects *sandboxed* programs (original Spectre-v1 [24]), they fail to protect constant-time programs, where secrets are loaded *non-speculatively* (see line 3 in Listing 1).

Recent Spectre defenses for constant-time programs extend prior solutions to protect non-speculative secrets as well [12, 14, 31, 43]. For example, SPT [12] extends the taint tracking mechanism of STT and assumes all data in registers and memory are tainted unless they leak during the non-speculative, sequential execution of the program which means they are declassified intentionally and can be untainted. Most proposals for constant-time programs increase the performance overhead compared to sandboxed cases since they have no other choice than restricting the execution for the instructions that are actually processing secrets, as seen in cryptographic applications. Our goal is to protect Spectre-type gadgets in cryptographic code, and to the best of our knowledge, our approach presented in this paper is the first defense that exploits key characteristics of constant-time cryptographic programs to improve performance compared to an unprotected baseline, while adhering to a strong sequential security guarantee.

3 THREAT MODEL

CASSANDRA exclusively protects Spectre-type gadgets in constant-time cryptographic code as the primary programs processing secrets. CASSANDRA does not provide protection for software isolation (i.e., sandboxing policy). Existing lightweight process isolation techniques (e.g., DyPrIs for clouds [45] and Site Isolation for browsers [38]) can prevent unintended transient leaks of non-crypto programs.

Meltdown-type attacks (e.g., Meltdown [29], LVI [53], Foreshadow [52], and MDS [6, 44, 54]) are out of scope. These attacks exploit the transient execution upon exceptions and CPU faults before they are handled, which are efficiently mitigated in recent CPUs via microcode updates [19].

4 BRANCH ANALYSIS OF CRYPTOGRAPHIC PROGRAMS

In this section, we investigate the practicality of a *record-and-replay* solution for cryptographic programs. to address control flow speculation. In §4.1, we discuss our key insights

enabling our proposed defense, and in §4.2, we detail our branch analysis.

4.1 Key Insights

We discuss two key insights from constant-time cryptographic programs.

> **Insight 1**: *Sequential control flow of constant-time cryptographic programs is a property of the algorithm and its implementation, and is known before execution.*

As we discussed in §2.1, constant-time principles assume that the entire control flow trace and memory addresses are leaked [2]. Hence, the dynamic control flow of the program is required to be independent from confidential inputs. On the other hand, public parameters of cryptographic programs are specified by standards or determined by the underlying scheme and its implementation, e.g., the key length, array sizes, encryption rounds, etc. As a result, the sequential and dynamic control flow of these programs is known before execution and does not change during runtime. This enables us to pre-compute sequential branch traces and enforce them during runtime, instead of using the BPU to predict the branch directions.

While branch traces of cryptographic programs can be computed before execution, they can still be huge and incur penalties to load them in the CPU. Our **Insight 2** enables us to significantly compress the branch traces; fitting the entire trace of most branches in a single entry of a small structure in the CPU.

> **Insight 2**: *Sequential control flow of constant-time cryptographic programs is highly regular and looped, allowing to significantly compress them.*

Most operations and transformations of constant-time cryptographic programs occur in loops (like Listing 1); Definition 1 allows one to wrap the operations in loops if the loop count is public. Hence, this insight enables us to detect the repeating patterns of each branch and only communicate this pattern with the CPU to repeatedly replay.

4.2 Detailed Branch Analysis

In this section, we investigate branches in different constant-time cryptographic programs from BearSSL library [3]. Note, that we consider all types of branches: conditional direct branches, unconditional indirect branches, returns, etc. To collect branch traces, we use Intel Pin [32] and dump the branch target at each execution of a branch. Figure 1 shows an overview of our branch analysis steps. As the first step (step ❶ in Figure 1), we collect the *raw* traces for each static branch. In this trace, we capture all the target PCs of a branch (i.e., the branch outcome) in the order they are executed (for
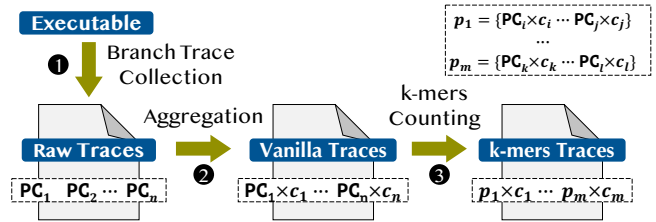

Figure 1: Branch analysis overview in CASSANDRA. Traces are per static branch.

not-taken cases, we dump the next PC). Here is an example of *raw* trace of a loop branch BR_0 with a loop count of four:

$$PC_1 \cdot PC_1 \cdot PC_1 \cdot PC_1 \cdot PC_0$$

where PC_1 is the taken path of the branch and PC_0 is the next PC after BR_0 (i.e., the not-taken path).

The next step of the analysis builds the *vanilla* traces which is a more compact format of the *raw* traces (step ❷). In this format, we aggregate the branch outcomes that are repeating and replace them with the repeated outcome PC and number of repetitions (i.e., its count). Here is the *vanilla* trace of branch BR_0 that we discussed earlier:

$$PC_1 \times 4 \cdot PC_0 \times 1$$

Vanilla traces are the baseline traces that we use for analysis and compression. Table 1 shows the average and maximum size of the *vanilla* traces in BearSSL programs. The results show that the average size of *vanilla* traces per branch is 384,947 and the maximum size is 51,538,410[1]. Communicating these large traces to the hardware can incur high efficiency overheads. However, we expect these traces to be represented by fewer elements according to **Insight 2**; we only need to detect the repeating outcome patterns of each static branch. We aim to devise a generic approach that can detect the repeating and unknown patterns in *vanilla* traces.

> QUESTION: How does one detect the repeating and unknown patterns and their frequency in a *vanilla* trace?

Detecting repeating, unknown patterns in large traces has been the focus of many domains, like database mining [1] and DNA sequencing [4, 33]. For example, two interesting problems in DNA sequencing are finding tandem repeats [4] and k-mers counting [33]. A tandem repeat in a DNA sequence is two or more contiguous copies of a pattern of nucleotides. Finding tandem repeats has applications like individual identification, tracing the root of an outbreak, etc. k-mers are also referred to a substring of size k of a given DNA sequence. Counting the frequency of k-mers is useful in genome assembly, sequence alignment, etc.

4.2.1 k-mers Counting and Traces. In this work, we deploy the k-mers counting technique for pattern repeat detection

---

[1]Here, size refers to the number of elements in a trace, not storage size.

Algorithm 1: *k-mers* Branch Compression

Input: DNA sequence *seq*, *max_k*
Output: *k-mers* trace K and pattern set P
1   $unused\_letters \leftarrow alphabet \setminus \mathrm{unique\_letters}(seq)$
2   $current\_len \leftarrow \infty$
3   **while**
    $\mathrm{len}(seq) < current\_len \land \mathrm{Size}(frequent\_kmers) < max\_k$
    do
4      $current\_len = \mathrm{len}(seq)$
5   **for** $k \leftarrow 2$ **to** max_k **do**
6        $freqs \leftarrow \mathrm{count\_kmers}(seq, k)$
7        $coverage.\mathrm{clear}()$
8   **foreach** $kmer \in freqs$ **do**
9   **if** $fres[kmer] > 1 \land \mathrm{Size}(kmer) \leq max\_k$ **then**
10            $coverage[kmer] \leftarrow$
            $(k \times freqs[kmer])/len(seq)$
11   **end**
12   **end**
13   **end**
14      $most\_frequent\_kmer \leftarrow \mathrm{max}(coverage)$
15      $frequent\_kmers.\mathrm{insert}(most\_frequent\_kmer)$
16      $letter \leftarrow unused\_letters.\mathrm{pop}()$
17      $seq.\mathrm{replace\_and\_merge}(most\_frequent\_kmer, letter)$
18   **end**
19   $K \leftarrow seq$
20   $P \leftarrow frequent\_kmers$

Table 1: Branch analysis of BearSSL programs. max_k **is set to 16 in our analysis. Note, that** *k-mers* **trace size is the sum of trace size and its pattern set size.**

Program	*Vanilla* trace size		*k-mers* trace size		*k-mers* compression rate	
	Avg	Max	Avg	Max	Avg	Max
RSA-2048	221,619.8	24,340,548	37.0	2,733	18,677.2	1,622,703.2
EC_c25519	965,261.6	51,538,410	7.9	134	321,607.7	17,179,470.0
DES	1,483,319.9	24,000,000	7.1	34	494,420.8	8,000,000.0
AES-128	161.5	1,530	6.5	31	43.8	510.0
ChaCha20	175.7	752	34.9	561	40.9	250.7
Poly1305	45.0	600	11.2	134	9.4	200.0
SHA-256	3,350.5	31,736	10.7	70	1,077.6	10,578.7
All	384,946.7	51,538,410	21.2	2,733	106,555.4	17,179,470.0

existing *k-mers* and their frequency. Algorithm 1 continues compressing the sequence with the most frequent pattern (i.e., has the highest coverage in the sequence, lines 14-17) until one of the two termination conditions occur (line 3):

(1) The length of the compressed sequence stops reducing;
(2) The size of the frequent patterns set has reached *max_k*.

Finally, the output of the algorithm is the compressed DNA sequence K and the set of detected patterns P (lines 19-20).

As the final step, we re-transform the DNA *k-mers* patterns back to the PC traces. We refer to the result as *k-mers* representation; *k-mers* representation consists of the *k-mers* trace K and its transformed pattern set P. For example, here is the *k-mers* trace of branch BR_1 that we discussed earlier:

$$p_0 \times 2 \cdot p_1 \times 1$$

where the pattern set is:

$$P = \{p_0 : PC_0 \times 2 \cdot PC_1 \times 5, p_1 : PC_2 \times 3\}$$

Table 1 shows the average and maximum size of *k-mers* representation (sum of trace K size and pattern set P size) for BearSSL programs. The average *k-mers* size per static branch is 21 and the maximum size is 2,733. Compared to *vanilla* trace sizes, our compression leads to an average compression rate of 106,555× and a maximum rate of 17,179,470×. Note, that the results presented in Table 1 exclude the branches that always have a single target (i.e., their *vanilla* trace size is already 1).

Example: Toy-AES-2. Figure 2 illustrates the Cassandra branch analysis for a toy example, Toy-AES-2 program, that encrypts data with key and plaintext length of two in three encryption rounds. As the first step, *raw* traces are collected per static branch (step ❶). For example, BR6 is a loop branch with a loop count of 2: it executes BR7 twice and then executes the fall-through path, PC7. In the next step, *vanilla* traces are generated (step ❷). After transforming *vanilla* traces into their equivalent DNA sequences (step ❸), we perform our *k-mers* branch compression technique and generate the *k-mers* traces and pattern sets (step ❹).

(step ❸). The reason for this choice is that our experiments with the state-of-the-art tools show that *k-mers* counting tools are much faster to analyze large traces (up to millions) compared to others (e.g., TRF tool [4] for tandem repeat finding) and also they are more configurable. We use scikit-bio Python library [46] in our analysis which allows us to define a custom alphabet for DNA sequences, while most tools only consider four letters A, C, G, T; some branches can have more than four outcomes (e.g., a return can jump to more than four callsites). Additionally, *k-mers* counting tools allow configuring the algorithm parameters which is useful to enforce starting with smaller and more frequent patterns and then continuing to larger patterns if necessary. This is beneficial to enable minimal storage.

Before *k-mers* counting, we transform *vanilla* traces to their equivalent DNA sequences. For example, *vanilla* trace of branch BR_1 of this form:

$$PC_0 \times 2 \cdot PC_1 \times 5 \cdot PC_0 \times 2 \cdot PC_1 \times 5 \cdot PC_2 \times 3$$

is transformed to this DNA sequence: *ACACG*.

Algorithm 1 shows a simplified version of the technique that we use to build *k-mers* traces. The inputs of the algorithm are (1) the equivalent DNA sequence of a *vanilla* trace and (2) *max_k* which specifies the maximum size of repeating patterns that we consider. The core of the algorithm is the *count_kmers* procedure (line 6) that takes k and DNA sequence *seq* as input and builds a frequency map of all the
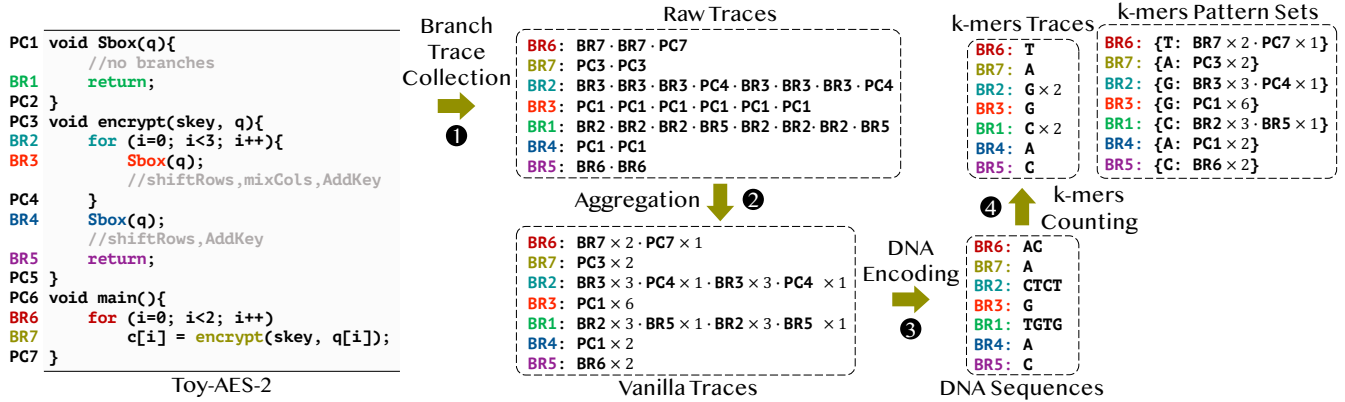
5

Figure 2: CASSANDRA branch analysis workflow example. Note, that the branches are analyzed separately and traces are generated per static branch; DNA sequences of branches are independent from each other.

Algorithm 2: Trace Generation Procedure

Input: Input binary bin_in, $inp1$, $inp2$
Output: Updated binary bin_out with traces and hint information

1 $traces$.clear()
2 $unique\_branches \leftarrow$ detect_static_branches($bin\_in, inp1$) **Ⓐ**
3 **foreach** $branch \in unique\_branches$ **do**
4      $[K_1, P_1] \leftarrow$ generate_kmers_traces($bin\_in, inp1$)
5      $[K_2, P_2] \leftarrow$ generate_kmers_traces($bin\_in, inp2$)
6      $is\_stream\_loop \leftarrow$ diff(K_1, K_2)
7   **if** $\neg is\_stream\_loop$ **then**
8         $traces$.insert($[branch, K_1, P_1]$)
9   **end**
10 **end**
11 $bin\_out \leftarrow$ embed_information($bin\_in, traces$)
12 **Procedure** generate_kmers_traces(bin, inp)
13      $R \leftarrow$ collect_raw_traces($bin\_in, inp1$) **Ⓑ**
14      $V \leftarrow$ transform_to_vanilla_traces(R) **Ⓒ**
15      $seq \leftarrow$ transform_to_DNA(V_1) **Ⓓ**
16      $[K, P] \leftarrow$ kmers_compression(DNA_seq, max_k) **Ⓔ**
17   **return** $[K, P]$

4.3 Trace Generation Procedure

In this section, we provide an automatic procedure to generate branch traces for a given binary of a constant-time cryptographic application. Algorithm 2 shows the steps of this procedure (steps **Ⓐ**-**Ⓔ**).

Step **Ⓐ** identifies all static branches that appear during the execution (line 2) and stores them in the *unique_branches* set. Steps **Ⓑ**-**Ⓔ** steps generate *k-mers* traces for each branch, as we explained in §4.2.

Note, that in lines 4 and 5, we generate *k-mers* traces twice to detect the *stream loop*. Stream ciphers, like ChaCha20, accept input plaintexts of an arbitrary length. The program processes each block of the plaintext in a loop (i.e., the stream loop). The *vanilla* trace of the stream loop is in the form of $PC_1 \times n \cdot PC_0 \times 1$, where n is the length of the input. However,

all the other branches are wrapped inside this loop and repeat. Hence, they have valid *k-mers* traces. For the stream loop, we stall the fetch until the stream loop resolves[2]; this incurs negligible penalty since it is not a long latency branch.

Finally, once all branches are analyzed, the input binary is instrumented with the *k-mers* traces and their *hint information* to facilitate their access during execution (line 13, see §5.2 for the details of trace representations and their communication with the hardware). We evaluate the analysis time of the trace generation procedure in §7.5.

5 DESIGN OF CASSANDRA

To implement CASSANDRA in hardware, we need to (1) communicate the branch traces prepared by our analysis with the hardware on demand, and (2) design a specialized unit, called *Branch Trace Unit* (BTU), in the fetch stage to determine the branch directions based on the branch traces. BTU is designed similarly to Trace Caches [39, 40] and Schedule Caches [35] in prior work, with two key differences: (1) traces are determined before execution in CASSANDRA and no dynamic trace selection methodology is required. (2) In case of a trace miss in the BTU, the frontend is stalled until the trace becomes available, while prior works would switch to a normal fetch procedure.

In §5.1, we present an overview of CASSANDRA design, and in §5.2 and §5.3, we provide the required details for CASSANDRA implementation.

5.1 Overview

Figure 3 shows an overview of the CASSANDRA microarchitecture. When a branch is fetched, two possible scenarios occur depending on whether the branch is accompanied by

---

[2]In general, if traces are not available for a crypto branch, we redirect fetch only if the branch direction is resolved.
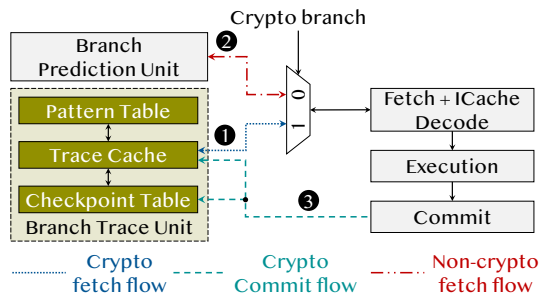
Figure 3: Overview of CASSANDRA microarchitecture. Crypto branches do not access or update the BPU.

Figure 4: Elements in the *Branch Trace Unit* (*BTU*). Each entry of the *Pattern Table, Trace Cache,* and *Checkpoint Table,* consisting of 16 elements and corresponds to a static branch.

our hint information (i.e., it is a crypto branch; see §5.2 for the details of branch hint information), or it is a non-crypto branch. In the former scenario, the fetch unit queries the *BTU* to determine the next PC (step ❶), and in the latter scenario, the BPU predicts the next PC (step ❷). *Pattern Table* and *Trace Cache* are the two sub-components of the *BTU* that (1) determine the next PC for each branch and (2) keep track of the progress in the trace. In cases that a trace fits in one entry of the *Trace Cache*, it will rotate to keep replaying the trace. However, if the trace does not fit in one entry then the head element of the entry is removed when the branch commits, and the entire entry shifts and prefetches the upcoming parts of the trace at the back (step ❸). Finally, when a branch misses in the *BTU*, one of the entries is evicted and a checkpoint of its progress is taken in the *Checkpoint Table*. This checkpoint allows to resume the execution of the evicted branch when it reappears in the future. In §5.3, we discuss the details of our microarchitecture.

5.2 Trace Representation and Communication

We use the output of Algorithm 1 to prepare the branch traces. Traces consist of two parts per static branch: (1) the pattern set built from the k-*mers* patterns P, which stores all the possible branch outcome patterns, and (2) the branch trace built from the k-*mers* trace K. Figure 4a shows the structure of each element in the pattern set. Each pattern element has a 12-bit target offset (the signed difference between the branch PC and the target PC) and the number of its repetitions (8-bit). In cases where the number of repetitions exceeds 8 bits, the element is duplicated in a way that the sum of the two elements is equal to the original number:

$$\delta(BR_0) \times 300 \rightarrow \delta(BR_0) \times 255 \cdot \delta(BR_0) \times 45$$

We use a compact form to store the patterns in cases where patterns overlap. For example, if two patterns in a trace are *ACT* and *CTA*, then the output pattern set is *ACTA*.

Figure 4b shows the structure of each element in the branch trace. The first two components, *pattern index* and
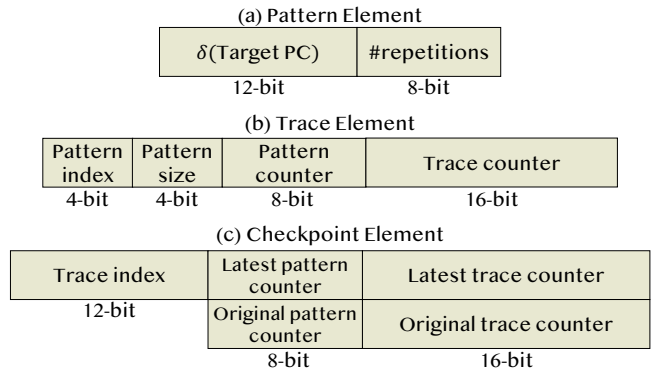
pattern size, specify the corresponding pattern from the pattern set. For example, if the corresponding pattern of a trace element is *CT* and the entire pattern set is *ACTA*, then the *pattern index* is 1 (indices start from 0) and the *pattern size* is 2. *Pattern counter* is equal to the sum of the repetitions of the corresponding pattern elements and the *trace counter* specifies the total number of times that the pattern needs to be repeated before advancing to the next trace element.

A special End of Trace marker is used to denote the end of each trace. This allows the processor to repeat the trace whenever it reaches the end of the trace. We store traces in data pages and embed hints for each static branch:

(1) *Single-target* mark. A significant portion of branches always jump to a single target (e.g., `"call sbox <pc>"`), and we mark such branches as *single-target* and do not need to store and communicate traces for them (e.g., 79% of static branches in RSA are *single-target*); we only need to embed its single target within the hint information (i.e., a *PC offset* pointing to the branch's single target). This implementation ensures that no *BTU* resources are used for *single-target* branches and no trace miss would occur as well.

(2) Traces Virtual Address offset (Δ). If the branch is *multiple-target*, then Δ points to the data page address that holds branch traces.

(3) *Short-trace* mark. We mark the branches when their traces are smaller than 16 (i.e., they fit in one entry of the *BTU*). This will allow us to reduce the memory accesses to bring traces to the core and only repeat the trace once loaded.

Embedding hint information. A general approach to inform the hardware about the hints is to insert a special hint instruction before each branch. Hint instructions are only decoded and do not use the ALUs; prior work has used hint instructions for x86 [22] and RISC-V [17]. An alternative solution is to re-purpose some of the previously-ignored prefix

bytes in x86, like how XRELEASE [30] was implemented, to embed the hint information for each branch, similar to [62]. Fourteen bits can be sufficient per static branch to embed single-target mark (1 bit), address offset (12 bits), and short-trace mark (1 bit). We opt to use the latter solution in this work because hint instructions still consume critical frontend resources, even though not executed. Moreover, inserting hint instructions might not provide backward compatibility with older processors. However, non-crypto branches do not need hint information and to avoid the penalties of waiting until the branch is decoded, one possible solution is to set new status registers that specify PC ranges for crypto code. This allows for early detection of crypto/non-crypto branches at the fetch stage (steps ❶ and ❷ in Figure 3).

5.3 Details of the Microarchitecture

The *BTU* consists of three main components:

- *Pattern Table* (*PAT*) holds the pattern sets of branches and each entry consists of 16 pattern elements (see Figure 4a), specified by *max_k* in Algorithm 1;
- *Trace Cache* (*TRC*) holds the branch traces and each entry consists of 16 trace elements (see Figure 4b);
- *Checkpoint Table* (*CPT*) always holds the latest valid position of the branch trace, i.e., the committed progress of the trace. Each entry is only one checkpoint element (see Figure 4c). *CPT* is stored in data pages which keeps the checkpoints for all branches to handle the *BTU* evictions and interrupts.

In addition, the *CPT* keeps the original counts of the first element of the *TRC* for (head of the trace); this helps the *BTU* to insert a refreshed version of the element at the back of the *TRC* entry for repetition (see the **commit flow** for the details of the *CPT* updates).

All three tables are direct-mapped tables, indexed with the branch PC, and they are fully inclusive of each other. The *BTU* uses an LRU replacement policy to evict an entry.

Crypto fetch flow. Once a crypto branch is fetched, the fetch unit queries the *BTU* to determine the next PC (step ❶ in Figure 3). If the branch is marked as *single-target*, then the next PC is already known by the hint information and there is no need for *BTU* lookup. For *multiple-target* branches, *BTU* looks up the first element of the *TRC* to find the appropriate pattern element in the *PAT* which provides the next PC. Upon each *BTU* lookup the *pattern counter* of the first element in the *TRC* is decremented. If the *pattern counter* is zero, then the *trace counter* is decremented and the *pattern counter* is refreshed based on the corresponding pattern elements. As we will explain in the **crypto commit flow**, the first element of the trace is removed only when the enforced branch direction is committed. Hence, there is a possibility that the *trace counter* of the first element is zero (i.e., we need to advance to the next element) but the branch is not committed yet. In this case, the *BTU* needs to lookup the next element in the *TRC* entry. In the worst case that all 16 elements of the *TRC* are looked up (i.e., *trace counter* is zero in all of them), then the *BTU* waits until the first element is removed. We did not encounter this scenario in our simulations since crypto branches are easy to resolve.

Non-crypto fetch flow. For non-crypto branches, we use the branch predictor to determine the next PC (step ❷).

Crypto commit flow. Once a crypto branch commits (step ❸), if the *trace counter* of the first element in the corresponding *TRC* entry is zero, then the first element is removed and all the other elements are shifted. If the branch is marked as *short-trace*, a refreshed version of the removed element is inserted at the back of the entry. However, if the trace is larger than the *TRC* entry, we prefetch the next element after the last element, which brings the upcoming elements to the *TRC* before they are needed. If the last element is an End of Trace marker, we restart from the beginning of the trace.

Additionally, when a crypto branch commits, the latest *pattern counter* and *trace counter* are checkpointed in the *CPT*. This allows the processor to resume the execution when it is interrupted (e.g., in context switches). *Trace index* in the checkpoint element (see Figure 4c) points to the latest trace element that the execution needs to resume from.

Trace evictions in the BTU. Once a trace is evicted from the *TRC*, the corresponding entries in the *PAT* and *CPT* are evicted as well. Before the eviction of the *CPT* entry, the checkpoint element is updated with the latest counters and *trace index* and is stored in the memory. This allows the CPU to resume the execution when the evicted branch reappears (this can commonly happen in context switches between different crypto applications that evict each others' traces).

Recovery for ROB Squashes. While CASSANDRA guarantees no branch mispredictions for crypto branches, ROB squashes can still occur due to other reasons (e.g., non-crypto mispredictions, interrupts, and exceptions), and CASSANDRA needs to recover in cases where the crypto branches are squashed. Whenever a crypto branch is squashed, we undo the actions of the **crypto fetch flow**; the *pattern counter* and *trace counter* of the first elements are incremented according to the checkpointed counters in the *CPT*.

5.4 Discussion

Q1: Is it safe to cache branch traces? *BTU* only contains the sequential control flow trace of the program; constant-time policies allow leaking this trace (i.e., it only depends on public information) and guarantee that it does not have any confidential information. Moreover, in §6, we formalize the *BTU* similar to caches.

Q2: Can non-crypto predictions cause security risks? Cases that non-crypto branches misspeculatively run non-crypto code is out of the scope of CASSANDRA's protection (see §3). However, we consider cases that which non-crypto branches misspuculatively redirect fetch to crypto code. Theoretically, BTB and RSB speculation primitives can jump to arbitrary locations, potentially to a crypto target, and force declassifying secrets before intended. However, exploiting this vulnerability is impractical in the CASSANDRA processor since such attacks usually require one to mistrain the BPU with legitimate code executions (i.e., crypto predictions) and later transiently execute through BTB collisions or RSB overflows. Since CASSANDRA never accesses or updates the BPU for crypto branches then it will never contain crypto targets. As a conservative measure, we envision performing an integrity check upon non-crypto predictions and avoid speculative fetch redirections if the predicted target is a crypto instruction, similar to CFI-informed speculations [26]. A global control bit can be set by the cryptographic programs to enable integrity checks when they start executing and disable it when the execution terminates (similar to the SSBD [20]).

Q3: Does CASSANDRA handle branches influenced by public parameters? As we discussed in §4.1, most crypto branches that are influenced by public parameters are specified by standards and underlying algorithms and do not change during execution. Hence, CASSANDRA would still generate traces for such branches. However, in some cases, different recommended modes exist for the same application (e.g., key sizes of 128, 192, and 256 for AES). One possible solution for these cases is generating separate traces for each mode and embedding all of them in the binary. A status register is set to specify the mode before execution and when combined with the hint information it allows to access the proper traces. An alternative solution can be to generate separate binaries for each mode.

Q4: Who will provide branch traces and when? Traces need to be re-generated after each compilation if PCs change. We believe developers can generate traces for recommended modes (e.g., AES-128/192/256) and embed hint information in binaries using our automated tool (§4.3). However, users can also generate traces for legacy binaries of cryptographic programs they intend to run on a CASSANDRA-enabled processor with the same procedure.

6 FORMAL SECURITY ANALYSIS

We provide a formalization of CASSANDRA on top of prior work [16] and express its formal security guarantees. Informally, we first choose a strong security contract and then ensure that the hardware semantics govern that all produced observations agree with the contract. We refer to this approach as *contract-informed hardware semantics*. While many works try to infer contracts for a new microarchitecture, we use contracts for a clean-slate design of our microarchitecture, starting with a strong contract. Our key observations from cryptographic programs and innovations in trace compression enable an efficient implementation of CASSANDRA's semantics.

§6.1 provides the background on hardware-software contracts [16] as our baseline framework. We specify CASSANDRA semantics in §6.2 and prove its security in §6.3.

6.1 Preliminaries on Hardware-Software Contracts

6.1.1 ISA Language. We rely on the μASM language, a small assembly-like language [16], with the following syntax:

(Expressions)	e	::=	$n \mid x \mid \ominus e \mid e_1 \otimes e_2$
(Instructions)	i	::=	$x \leftarrow e \mid \textbf{load } x, e \mid \textbf{store } x, e$
			$\mid \textbf{call } f \mid \textbf{beqz } x, \ell \mid \textbf{ret}$
(Functions)	$\mathscr{F}$	::=	$\varnothing \mid \mathscr{F}; f \mapsto n$
(Crypto Tags)	t	::=	$c \mid \varepsilon$
(Programs)	p	::=	$i@t \mid p_1; p_2$

where $x \in Regs$ and $n, \ell \in Vals = \mathbb{N} \cup \{\bot\}$. The $\textbf{pc} \in Regs$ refers to a special register that contains the program counter. In addition, an architectural state $\sigma = \langle m, a \rangle$ consists of the memory $m : Vals \rightarrow Vals$, and register assignment $a : Regs \rightarrow Vals$. Each instruction has a tag t that specifies if it is a crypto instruction and analyzed by CASSANDRA; crypto instructions are tagged as c and the rest are untagged (i.e., ε).

6.1.2 Contracts. A contract governs the attacker-visible observations of a given program. A contract $[\![\cdot]\!]_\beta^\alpha$ has two main components:

- *Execution model α* specifies how state transitions occur. For example, the sequential model (denoted as seq) evaluates the branch condition before transitioning to the next state, while a speculative model (denoted as spec) predicts the target.
- *Leakage model β* specifies the leakages that are observable by an attacker. For example, the constant-time leakage model (denoted as ct) leaks the control flow and memory addresses.

Contract semantics $\overset{\tau_n}{\rightsquigarrow}$ is labeled with the *observations* τ_n when transiting between two architectural states. Observations τ_i capture leaks via cache and control flow:

$$
\begin{aligned}
CfObs &::= \textsf{pc } n \mid \textsf{call } f \mid \textsf{ret } n \\
MemObs &::= \textsf{load } n \mid \textsf{store } n \\
Obs &::= MemObs \mid CfObs \\
\tau ::= \varepsilon \mid Obs \quad &\bar{\tau} ::= \emptyset \mid \bar{\tau} \cdot \tau@t
\end{aligned}
$$

The `pc n`, `call f`, and `ret n` observations record the control flow of the program (denoted as *CfObs*). The `load n` and `store n` observations record the memory addresses to capture cache leakage (denoted as *MemObs*). In addition, observations are tagged with the same crypto tag of the instruction that generates the observation.

For a given program p and initial architectural state σ_0, the labels of the transitions in run $\sigma_0 \overset{\tau_1@t_1}{\rightsquigarrow} \sigma_1 \overset{\tau_2@t_2}{\rightsquigarrow} \ldots \overset{\tau_n@t_n}{\rightsquigarrow} \sigma_n$ produce the contract trace $[\![p]\!](\sigma_0) = [\tau_1@t_1 \cdot \ldots \cdot \tau_n@t_n]$.

Contract $[\![\,\cdot\,]\!]_{\text{ct}}^{\text{seq}}$. This contract specifies the strongest security guarantee for secure speculation mechanisms (i.e., sequential execution model for constant-time leakages). For example, two rules of $[\![\,\cdot\,]\!]_{\text{ct}}^{\text{seq}}$ contract are as follows:

(Beqz-Sat)
$$\frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, \ell@t \qquad \langle m, a \rangle \rightarrow \langle m', a' \rangle}{\langle m, a \rangle \overset{\text{pc}\ \ell@t}{\rightsquigarrow} \langle m', a' \rangle}$$

(Load)
$$\frac{p(a(\mathbf{pc})) = \mathbf{load}\ x, e@t \qquad \langle m, a \rangle \rightarrow \langle m', a' \rangle}{\langle m, a \rangle \overset{\text{load}\ n@t}{\rightsquigarrow} \langle m', a' \rangle}$$

where $n = [\![e]\!](a)$ is the result of expression e given register assignment a. $[\![\,\cdot\,]\!]_{\text{ct}}^{\text{seq}}$ exposes the control flow (`pc n`, `call f`, and `ret n`) and memory addresses (`load n` and `store n`) in a sequential execution model. Note, that the values of loads and stores are not leaked.

6.2 Cassandra Semantics

In this section, we formalize a contract-informed semantics for the Cassandra methodology (denoted as $\{\!\{\,\cdot\,\}\!\}_{\text{csd}}$).

As the first step, we define auxiliary contract traces to enable our contract-informed hardware semantics:

Definition 2 (Crypto control flow trace $\mathscr{C}$). For a given program p, initial architectural state σ_0 and contract $[\![\,\cdot\,]\!]_{\beta}^{\alpha}$, $\mathscr{C}_{\beta}^{\alpha}(p, \sigma_0)$ is a subtrace of contract trace $[\![p]\!]_{\beta}^{\alpha}(\sigma_0) = [\tau_1@t_1 \cdot \ldots \tau_n@t_n]$, consisting of all crypto control flow observations:
$$\mathscr{C}_{\beta}^{\alpha}(p, \sigma_0) = [\tau_i@c\,|\,1 \leq i \leq n, \tau_i \in CfObs]$$

We can define the contract memory trace $\mathscr{M}_{\beta}^{\alpha}(p, \sigma_0)$ in the same way where it consists of only memory observations. Since we target constant-time cryptographic programs, the $\mathscr{C}_{\beta}^{\alpha}(p, \sigma_0)$ trace is independent from σ_0, and we use $\mathscr{C}_{\beta}^{\alpha}(p)$ for brevity. Note, that $\mathscr{C}_{\beta}^{\alpha}(p)(i)$ refers to the ith observation of the crypto control flow trace of contract $[\![\,\cdot\,]\!]_{\beta}^{\alpha}$.

Hardware configuration. Hardware configuration ω in Cassandra consists of (1) the architectural state σ with the memory m and register assignment a, (2) a global counter $\mathbf{C}$ that counts the number of contract-level control flow observations, (3) reorder buffer *buf* with size $\mathbf{B}$, and (4) the microarchitectural context μ. Microarchitectural context is

Table 2: Signatures of Cassandra microarchitecture.

Component	States	Functions
Cache	*CacheStates*	access : *CacheStates* × *Vals* → {Hit, Miss}
		update : *CacheStates* × *Vals* → *CacheStates*
Scheduler	*ScStates*	next : *ScStates* → {Fetch, Execute, Commit}
		update : *ScStates* × *Bufs* → *ScStates*
Trace Cache	*TcStates*	access : *TcStates* × *Vals* → {Hit, Miss}
		update : *TcStates* × *Vals* → *TcStates*

the part of the microarchitecture that the attacker can observe or influence. We use an abstract model for caches and pipeline scheduler, similar to [16], and also add the trace cache (specific to the Cassandra semantics, representing the *BTU*). Table 2 shows the interface of each component. For simplicity, we do not include the branch predictor since it is not accessed or influenced in our semantics.

- *Cache*: The access function results in a Hit or Miss based on a given cache state cs and memory address ℓ and the update function generates a new cache state based on the input cache state and memory address;
- *Scheduler*: The next function determines the next processor step (Fetch, Execute, or Commit) given the scheduler state sc, and the update function updates the scheduler's state based on the reorder buffer state;
- *Trace Cache*: The access function results in a Hit or Miss based on a given trace cache state tc and branch PC ℓ. The update function updates the trace cache state if needed (e.g., fetching traces for a given branch PC ℓ that misses in the trace cache).

Moreover, a *reorder buffer* records the state of in-flight instructions. Expressions in a reorder buffer are initially unresolved and they can transform to a resolved state after execution. A data-independent projection of a reorder buffer is shown as $buf\downarrow$ where resolved expressions are replaced with R and unresolved expressions with UR. In addition to [16], we define an examine function that outputs R for a given $buf\downarrow$ if all expressions are resolved:

$$\text{examine}(buf\downarrow) = \begin{cases} \text{R} & \text{if all expressions in } buf\downarrow \text{ are R} \\ \text{UR} & \text{otherwise} \end{cases}$$

Cassandra semantics uses a binary relation ($\rightarrowtail_{\text{csd}}$) that maps hardware configurations to their successors:

(Step-Cassandra)
$$\frac{\begin{array}{cc} d = \text{next}(sc) & sc' = \text{update}(sc, buf') \end{array}}{\langle m, a, \mathbf{C}, buf, cs, tc \rangle \overset{d}{\rightarrow} \langle m', a', \mathbf{C}', buf', cs', tc' \rangle}{\langle m, a, \mathbf{C}, buf, cs, tc, sc \rangle \ \rightarrowtail_{\text{csd}}\ \langle m', a', \mathbf{C}', buf', cs', tc', sc' \rangle}$$

Given the current hardware state $\omega = \langle m, a, \mathbf{C}, buf, cs, tc, sc \rangle$, the rule Step-Cassandra finds the next directive d via the next(sc) and takes an appropriate step (formalized through the fetch, execution, and commit rules) that produces the new state $\omega' = \langle m', a', \mathbf{C}', buf', cs', tc', sc' \rangle$.

Most transition rules of CASSANDRA are standard and the same as the baseline in [16] and not presented here for brevity. The difference of the CASSANDRA rules compared to the baseline occurs in the fetch stage, when a branch hits in the cache. While the baseline semantics use the FETCH-BRANCH-HIT rule to determine the fetch direction via prediction, CASSANDRA replaces this with a new set of rules that determines the next PCs based on the crypto tags and the $[\![\cdot]\!]_{\mathrm{ct}}^{\mathrm{seq}}$ contract. For tagged branches, CASSANDRA uses the contract traces to determine the fetch direction. The first rule handles the case that the branch traces miss in the *Trace Cache* (our additions are highlighted):

(FETCH-BRANCH-HIT-TAGGED-TRACE-MISS)
$$a' = \mathsf{apl}(buf, a)$$
$$i = a'(\mathbf{pc}) \qquad p(i) = \mathbf{beqz}\ x, \ell@c \mid \mathbf{call}\ f@c \mid \mathbf{ret}@c$$
$$|buf| < \mathbf{B} \qquad \mathsf{access}(cs, i) = \mathsf{Hit} \qquad \mathsf{update}(cs, i) = cs'$$
$$\mathsf{access}(tc, i) = \mathsf{Miss} \qquad \mathsf{update}(tc, i) = tc'$$
$$\overline{\langle m, a, \mathbf{C}, buf, cs, tc \rangle \xrightarrow{\mathsf{fetch}} \langle m, a, \mathbf{C}, buf, cs', tc' \rangle}$$

In this rule, the *Trace Cache* is updated to bring the missed traces to hit later[3]. The second rule handles the case that branch traces hit in the *Trace Cache*:

(FETCH-BRANCH-HIT-TAGGED-TRACE-HIT)
$$a' = \mathsf{apl}(buf, a)$$
$$i = a'(\mathbf{pc}) \qquad p(i) = \mathbf{beqz}\ x, \ell@c \mid \mathbf{call}\ f@c \mid \mathbf{ret}@c$$
$$|buf| < \mathbf{B} \qquad \mathsf{access}(cs, i) = \mathsf{Hit}$$
$$\mathsf{update}(cs, i) = cs' \qquad \mathsf{access}(tc, i) = \mathsf{Hit}$$
$$\mathsf{update}(tc, i) = tc' \qquad \ell' = \mathscr{C}_{\mathrm{ct}}^{\mathrm{seq}}(p)(\mathbf{C})$$
$$\overline{\langle m, a, \mathbf{C}, buf, cs, tc \rangle \xrightarrow{\mathsf{fetch}} \langle m, a, \mathbf{C} + 1, buf \cdot \mathbf{pc} \leftarrow \ell', cs', tc' \rangle}$$

In this rule, the next PC is determined through contract-level observations.

While prior rules handle branches that are tagged by CASSANDRA, the remaining branches (e.g., the stream loop and non-crypto branches) need to be handled differently. Here, we assume the fetch is stalled until all instructions in the reorder buffer are resolved to enforce the $[\![\cdot]\!]_{\mathrm{ct}}^{\mathrm{seq}}$ contract (we use the examine function for this purpose). This choice is to guarantee safe interactions between crypto and non-crypto codes, however, processors can deploy more efficient solutions with $[\![\cdot]\!]_{\mathrm{ct}}^{\mathrm{seq}}$ guarantees to handle non-crypto branches. In §7.3, we explore the design idea of combining CASSANDRA and ProSpeCT [14].

---

[3]In all rules, $\mathsf{apl}(buf, a)$ obtains the new register assignment a' after applying the changes of resolved instructions in buf [16].

(FETCH-BRANCH-HIT-UNTAGGED-UNRESOLVED)
$$a' = \mathsf{apl}(buf, a)$$
$$i = a'(\mathbf{pc}) \qquad p(i) = \mathbf{beqz}\ x, \ell@\varepsilon \mid \mathbf{call}\ f@\varepsilon \mid \mathbf{ret}@\varepsilon$$
$$|buf| < \mathbf{B} \qquad \mathsf{access}(cs, i) = \mathsf{Hit}$$
$$\mathsf{update}(cs, i) = cs' \qquad \mathsf{examine}(buf\downarrow) = \mathsf{UR}$$
$$\overline{\langle m, a, \mathbf{C}, buf, cs, tc \rangle \xrightarrow{\mathsf{fetch}} \langle m, a, \mathbf{C}, buf, cs', tc' \rangle}$$

Once the reorder buffer is resolved, we determine the next PC to fetch based on the specific branch we are handling. Here, we only show a selected rule for conditional branches. The rules for calls and returns also use resolved information to find the next, sequential direction.

(FETCH-BRANCH-HIT-UNTAGGED-RESOLVED-BEQZ)
$$a' = \mathsf{apl}(buf, a) \qquad i = a'(\mathbf{pc}) \qquad p(i) = \mathbf{beqz}\ x, \ell@\varepsilon$$
$$|buf| < \mathbf{B} \qquad \mathsf{access}(cs, i) = \mathsf{Hit} \qquad \mathsf{update}(cs, i) = cs'$$
$$\mathsf{examine}(buf\downarrow) = \mathsf{R} \qquad \ell' = \begin{cases} i + 1 & \text{if } a'(x) \neq 0 \\ \ell & \text{if } a'(x) = 0 \end{cases}$$
$$\overline{\langle m, a, \mathbf{C}, buf, cs, tc \rangle \xrightarrow{\mathsf{fetch}} \langle m, a, \mathbf{C}, buf \cdot \mathbf{pc} \leftarrow \ell', cs', tc' \rangle}$$

Since CASSANDRA exploits contract-level observations of the $[\![\cdot]\!]_{\mathrm{ct}}^{\mathrm{seq}}$, it is guaranteed that no branch mispredictions happen and there is no need to recover the $\mathbf{C}$ state for squashing branches. In addition, we assume that data flow speculation is disabled and they cannot cause squashes as well (our experiments and prior work [34] show that disabling or naively addressing data flow speculation for cryptographic programs incurs negligible overheads).

6.3 Definitions and Theorems

Adversary model. We define the adversary as a projection function $\mathscr{A}$ that specifies observations from a microarchitectural context. For a given hardware semantics $\{\!|\cdot|\!\}$ and program p, hardware run $\omega_0 \rightarrowtail \omega_1 \rightarrowtail \ldots \rightarrowtail \omega_n$ produces the hardware observations: $\{\!|p|\!\}(\sigma_0) = [\mathscr{A}(\omega_0)\mathscr{A}(\omega_1)\ldots\mathscr{A}(\omega_n)]$.

Definition 3 ($\omega \approx \omega'$). Two hardware configurations $\omega = \langle m, a, \mathbf{C}, buf, cs, tc, sc \rangle$ and $\omega' = \langle m', a', \mathbf{C}, buf', cs', tc', sc' \rangle$ are *indistinguishable*, iff $\mathscr{A}(\omega) = \mathscr{A}(\omega')$.

We consider an adversary that observes the entire microarchitectural context, including the reorder buffer, the cache (which only contains the addresses, not the values), the trace cache, and the branch predictor.

To express security guarantees of a hardware semantics $\{\!|\cdot|\!\}$ against a contract $[\![\cdot]\!]$, we use Definition 4 [16].

Definition 4 ($\{\!|\cdot|\!\} \vdash [\![\cdot]\!]$). A hardware semantics $\{\!|\cdot|\!\}$ satisfies a contract $[\![\cdot]\!]$ if for an arbitrary program p and arbitrary initial architectural states σ, σ':
$$[\![p]\!](\sigma) = [\![p]\!](\sigma') \Rightarrow \{\!|p|\!\}(\sigma) = \{\!|p|\!\}(\sigma').$$

Note, that we require the initial microarchitectural components be the same for this definition.

THEOREM 1. $\{ \cdot \}_{csd} \vdash [\![\cdot]\!]_{ct}^{seq}$.

PROOF. Let p be an arbitrary program. Moreover, let $\sigma_0 = \langle m, a \rangle$ and $\sigma_0' = \langle m', a' \rangle$ be two arbitrary initial architectural states. Two possible cases are:

(1) $[\![p]\!]_{ct}^{seq}(\sigma_0) \neq [\![p]\!]_{ct}^{seq}(\sigma_0')$: which trivially holds $[\![p]\!](\sigma_0) = [\![p]\!](\sigma_0') \Rightarrow \{p\}(\sigma_0) = \{p\}(\sigma_0')$.

(2) $[\![p]\!]_{ct}^{seq}(\sigma_0) = [\![p]\!]_{ct}^{seq}(\sigma_0')$: By unrolling $[\![p]\!]_{ct}^{seq}(\sigma)$, two contract runs are obtained that agree on all observations ($\forall 0 \leq i \leq n : \tau_i = \tau_i'$):

$$\text{cr} := \sigma_0 \overset{\tau_1}{\rightsquigarrow} \sigma_1 \overset{\tau_2}{\rightsquigarrow} \ldots \overset{\tau_n}{\rightsquigarrow} \sigma_n \quad \text{cr}' := \sigma_0' \overset{\tau_1'}{\rightsquigarrow} \sigma_1' \overset{\tau_2'}{\rightsquigarrow} \ldots \overset{\tau_n'}{\rightsquigarrow} \sigma_n'$$

and produced hardware runs by $\{p\}_{csd}(\sigma_0)$ and $\{p\}_{csd}(\sigma_0')$ are:

$$\mathbf{hr} := \omega_0 \rightarrowtail_{csd} \omega_1 \rightarrowtail_{csd} \ldots \rightarrowtail_{csd} \omega_m$$
$$\mathbf{hr}' := \omega_0' \rightarrowtail_{csd} \omega_1' \rightarrowtail_{csd} \ldots \rightarrowtail_{csd} \omega_m'$$

where $\mathbf{hr}(i) = \omega_i$ and $\text{cr}(i) = \sigma_i$. We prove by induction that $\{p\}_{csd}(\sigma_0) = \{p\}_{csd}(\sigma_0')$, i.e., $\forall 0 \leq i \leq m : \mathbf{hr}(i) \approx \mathbf{hr}'(i)$.

(*Induction basis*): the initial hardware configurations $\mathbf{hr}(0)$ and $\mathbf{hr}'(0)$ are indistinguishable by definition as they agree on their microarchitectural components.

(*Inductive step*): assume that after i steps in $\{p\}_{csd}$: $\mathbf{hr}(i) \approx \mathbf{hr}'(i)$. Since in our hardware semantics observations are either informed by the contract $[\![\cdot]\!]_{ct}^{seq}$ (for tagged branches) or determined based on non-speculative, sequential information (for untagged branched), the corresponding contract runs of cr and cr' take the same steps k. In other words, the corresponding contract state of $\mathbf{hr}(i)$ is $\text{cr}(k)$, and the corresponding contract state of $\mathbf{hr}'(i)$ is $\text{cr}'(k)$. Based on our assumptions, (a) $\mathbf{hr}(i)$ and $\mathbf{hr}'(i)$ agree on all microarchitectural components and (b) the next steps of $\{p\}_{csd}$ to obtain $\mathbf{hr}(i+1)$ and $\mathbf{hr}'(i+1)$ are determined by the $\text{cr}(k+1)$ and $\text{cr}'(k+1)$ observations, which are the same by assumption. Hence, based on (a) and (b): $\mathbf{hr}(i+1) \approx \mathbf{hr}'(i+1)$. □

It is interesting to note that Theorem 1 and its proof are direct result of contract-informed semantics of $\{ \cdot \}_{csd}$, which ensure the hardware is secure by design.

7 EVALUATION

7.1 Experimental Setup

Simulation. We implement the CASSANDRA on top of the gem5 OoO core implementation and evaluate the design using gem5's Syscall Emulation (SE) mode. Table 3 shows the evaluated system configuration. We use a Golden-Cove-like microarchitecture [37]. We use McPAT 1.3 [27] and CACTI 6.5 [28] to investigate the power and area impacts.

Table 3: gem5 configuration for simulation.

Pipeline	8 F/D/I/C width, 192/114 LQ/SQ entries, 512 ROB entries, 96 IQ entries, 280/332 RF (INT/FP), 16 MSHRs, LTAGE branch predictor
BTU	16 *PAT*/*TRC*/*CPT* entries
L1 DCache	32 KB, 64 B line, 8-way, 2-cycle latency
L1 ICache	32 KB, 64 B line, 4-way, 2-cycle latency
L2 Cache	256 KB, 64 B line, 16-way, 20-cycle latency
L3 Cache	2 MB, 64 B line, 16-way, 40-cycle latency
DRAM	50 ns latency after L2

Workloads. We use the test applications from the BearSSL library [3]. For the applications with more than 1B instruction count, we used SimPoint methodology [47] to generate representative regions for realistic and practical simulation time-frames (an average of 6 SimPoints per application and 50M instructions per region was seen with our workloads). In §7.3, we also evaluate the SpectreGuard [15] synthetic benchmarks that are a mix of crypto and non-crypto code. Moreover, we used gem5 itself to collect branch traces for CASSANDRA, however, other tools can be used as well (e.g., Intel Pin [32] and DynamoRIO [5]).

7.2 BearSSL Performance Results

We evaluate four different designs in this section:

- *Unsafe Baseline*: unprotected baseline OoO processor, vulnerable to control flow and data flow speculation;
- CASSANDRA: our proposed design; addressing control flow speculation;
- CASSANDRA+*STL*: an extension of CASSANDRA that addresses data flow speculation as well; it always sends a request to memory even if there is a load-store address match, and also restricts the dependents of bypassing loads until prior stores resolve, similar to prior work [12, 31];
- *SPT*: a prior hardware-level defense [12]. We use the proposed setting of the work for the Spectre attack model.

Figure 5 shows the execution time of the BearSSL applications with different designs. CASSANDRA *improves* performance compared to the *Unsafe Baseline* by 1.77% on average. This is mainly because of the elimination of branch mispredictions, and as a result, no ROB squashes due to branch misprediction occur in our implementation.

In addition, the results show that extending CASSANDRA to protect data flow speculation (i.e., CASSANDRA+*STL*) achieves a performance improvement of 1.35%. Prior work also shows that setting the SSBD control bit in existing CPUs introduces negligible slowdown (less than 1%), probably due to easy-to-resolve address calculations in crypto primitives [34].
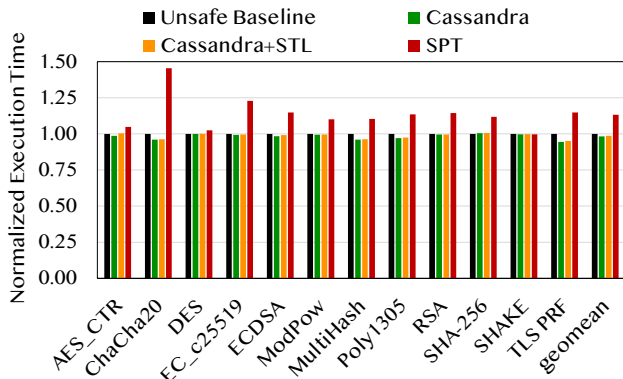
Figure 5: Execution time of different designs normalized to the *Unsafe Baseline*. A longer bar means higher overhead.

Finally, *SPT* shows a 13.27% performance overhead compared to the *Unsafe Baseline*, and a 15.31% overhead compared to the Cassandra. *SPT* has low overheads for some applications (e.g., 2.5% for DES), but can be significantly higher (up to 45.4%) for applications like ChaCha20, while Cassandra improves performance by 3.9% for ChaCha20 compared to the *Unsafe Baseline*.

7.3 Synthetic Benchmark Performance Results

In this section, we evaluate the synthetic benchmark from SpectreGuard [15], which is a mix of non-crypto, (s)andboxed code, and (c)rypto code (s/c indicates the fraction of each part). We evaluate two designs: (1) *ProSpeCT* [14], the state-of-the-art defense for constant-time programs, and (2) Cassandra+*ProSpeCT*. Note, that Cassandra only protects crypto branches and needs to be combined with *ProSpeCT* to protect non-crypto mispredictions. We use the benchmark and crypto primitives open-sourced by *ProSpeCT* authors with precise annotation of secret and public variables[4].

We implement *ProSpeCT* in gem5 and block execution under two conditions: (1) the instruction is speculative (i.e., there is an older, unresolved control inducer), and (2) the instruction is about to process a secret (i.e., one or more operands are tainted). Destination registers of loads from secret memory regions are taint sources that are propagated during execution. All registers are declassified (i.e., untainted) at the end of crypto primitives. Combining Cassandra with *ProSpeCT* is straightforward; we do not consider crypto branches as control inducers since their direction is specified by non-speculative, sequential information.

Figure 6 shows the performance impacts of *ProSpeCT* and Cassandra+*ProSpeCT* for two settings, running different

---
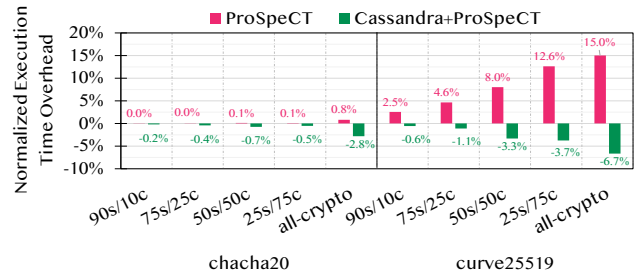[4]https://github.com/proteus-core/prospect/

Figure 6: Execution time of *ProSpeCT* and Cassandra+*ProSpeCT* for synthetic benchmark, normalized to the respective *Unsafe Baseline* of each configuration. Negative numbers mean performance improvement. In `chacha20` the stack is marked as public, while it is secret in `curve25519`.

primitives for the crypto component (HACL* chacha20 [63] and curve25519 [13]). For `chacha20`, *ProSpeCT* incurs no overhead for all combinations, and Cassandra shows marginal performance improvements. In this setting, the overall performance is dominated by the non-crypto component which limits the benefits of Cassandra. We evaluate the case where only the crypto primitive runs (`all-crypto`, similar to Figure 5). Cassandra+*ProSpeCT* shows a 2.8% performance improvement in this case, and *ProSpeCT* incurs negligible overhead of 0.8%.

For `curve25519`, *ProSpeCT* marks the stack as secret, unlike `chacha20`. Curve25519 is more complex and it is not trivial to manually avoid spilling secrets to the stack. In this case, *ProSpeCT* can see slowdowns. Additionally, `curve25519` is more control intensive compared to `chacha20` and has higher impact on the overall performance. Figure 6 shows that *ProSpeCT* incurs an overhead between 2.5% and 12.6% when increasing the crypto component from `10c` to `75c`. This overhead reaches its peak when only crypto code runs (15.0% for `all-crypto`). Interestingly, Cassandra can relax *ProSpeCT*'s restrictions for the crypto code and improve performance by increasing the crypto component (0.6% for `90s/10c` and 3.7% for `25s/75c`). Finally, Cassandra improves performance for `all-crypto` by 6.7%. Note, that *ProSpeCT* is still enabled to prevent unintended leaks due to non-crypto mispredictions. The main reasons for Cassandra's improvements are: (1) Cassandra can relax most of *ProSpeCT*'s restrictions due to crypto branches, and (2) it eliminates the penalties of crypto branch mispredictions and squash cycles.

7.4 Power and Area Impacts

Figure 7 shows power consumption and area of Cassandra compared to *Unsafe Baseline*. The results show that Cassandra is able to reduce the power consumption compared to the *Unsafe Baseline* by 2.73%. The main reason is that crypto branches avoid accessing and updating the BPU and

Table 4: Analysis time for trace generation of BearSSL programs (see Algorithm 2 for the details of each step). Numbers are in seconds. *Note, that we exclude the branches that have a single target.

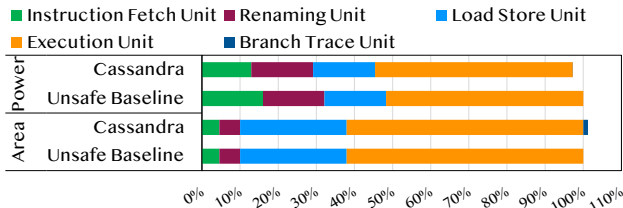| Program	#static branches*	Branch detection Ⓐ	*Raw* trace collection Ⓑ		*Vanilla* trace transformation Ⓒ		DNA transformation Ⓓ		*k-mers* compression Ⓔ	
			Avg	Max	Avg	Max	Avg	Max	Avg	Max
RSA-2048	159	631.170	6.609	88.171	0.625	31.578	0.214	11.742	3.115	144.881
EC_c25519	83	914.978	13.280	320.508	1.509	72.852	0.539	27.502	6.411	290.633
DES	53	1164.927	76.872	713.818	5.160	72.429	0.743	11.589	9.476	138.620
AES-128	27	0.390	0.497	0.540	0.041	0.051	0.041	0.048	0.964	1.099
ChaCha20	24	0.737	0.452	0.498	0.041	0.044	0.040	0.043	0.939	1.010
Poly1305	28	0.462	0.446	0.469	0.041	0.048	0.041	0.042	0.927	0.953
SHA-256	30	4.098	0.479	0.498	0.051	0.210	0.042	0.056	0.957	1.129
Average	57.7	388.109	14.091	160.643	1.067	25.316	0.237	7.289	3.256	82.618

Figure 7: Power and area of Cassandra, normalized to the total power and area of the *Unsafe Baseline*.

access *BTU* as a smaller and simpler unit (see the reduction in *Instruction Fetch Unit*). Our results confirm that Cassandra will not add power overheads, nevertheless, the benefits might not be as high when combined with non-crypto workloads. Finally, *BTU* has an area overhead of 1.26%.

7.5 Upfront Trace Generation Runtime Overhead

Table 4 shows the analysis time for each step of the trace generation procedure (steps Ⓐ-Ⓔ in Algorithm 2) for BearSSL programs. We use Intel Pin [32] for dynamic analysis and gathering *raw* traces. Branch detection (step Ⓐ) is executed once per application and it takes 388 seconds on average (i.e., 6 minutes and 28 seconds). Steps Ⓑ-Ⓔ are executed per static branch. The results show that collecting *raw* traces (step Ⓑ) takes 14 seconds on average per branch and *k-mers* compression (step Ⓔ) takes about 3 seconds on average.

8 RELATED WORK

Hardware-only Spectre defenses. Prior works have investigated hardware defenses to protect constant-time programs [12, 31]. These defenses are complex to design as they need to track speculation taints in all potential microarchitectural components which can also incur high performance overheads due to limited knowledge about the running applications and their security policy. Cassandra only adds a small structure (i.e., *BTU*), which has better performance and

less power compared to the baseline, with modest changes in the microarchitecture.

Software-only Spectre defenses. To harden programs on existing CPUs, compiler passes were designed that take the speculative execution model of CPUs into account and insert protections as needed [11, 49, 61]. However, software-level defenses can result in prohibitive slowdowns.

Hardware/software co-designed defenses. Similar to Cassandra, some prior defenses require the cooperation of both hardware and software [14, 15, 34, 43, 50, 58]. Serberus [34] is the state-of-the-art compiler mitigation that addresses all speculation primitives on existing hardware. Serberus shows different slowdowns depending on the cipher buffer size (21% slowdown for small buffers of 64B and 8% for large buffers of 8KB). For our results, a buffer size of 4KB is used in the synthetic benchmark and the default buffers of BearSSL tests are used (e.g., ChaCha20 uses a buffer size of 400B) [3]. The performance gap between Serberus and Cassandra will be smaller for larger buffers. ProSpeCT [14] is the state-of-the-art defense for future hardware that requires manual secret annotations in the program and blocking the execution for transient instructions that process secrets. We provide a detailed performance comparison with Cassandra in §7.3. Additionally, ProSpeCT reports a 17% increase in the number of slice LUTs and a 2% increase for the critical path when their hardware is synthesized for an FPGA [14].

Profile-guided branch analysis. There have been studies to use runtime profiles of applications to eliminate branch mispredictions [21, 22, 60]. These techniques mainly target data center applications since they have large code footprints and frequent branch mispredictions. For example, Whisper [22] proposes a profile-guided approach that provides hints per static branch to help the branch predictor avoid mispredictions. However, the goal of these solutions is to build *approximately* accurate branch histories, but still rely on the branch predictor to steer the fetch direction.

9   CONCLUSION

In this work, we propose Cassandra, a novel hardware-software mechanism to protect constant-time cryptographic programs against speculative execution attacks. To achieve this, we perform an offline branch analysis step to significantly compress sequential branch traces and communicate them with the hardware. During execution, the processor uses the branch traces to determine fetch directions and to avoid accessing the branch predictor. Moreover, we formalize and prove the security of Cassandra by introducing contract-informed hardware semantics that ensures the hardware adheres to a strong security contract by design. Despite providing a high security guarantee, Cassandra counterintuitively improves performance by 1.77%.

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*, pages 3–14. IEEE, 1995.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, 2016.

[3] BearSSL - constant-time crypto library. https://www.bearssl.org, Accessed 22-11-2023.

[4] Gary Benson. Tandem repeats finder: a program to analyze dna sequences. *Nucleic acids research*, 27(2):573–580, 1999.

[5] Derek Bruening and Saman Amarasinghe. Efficient, transparent, and comprehensive runtime code manipulation. 2004.

[6] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, 2019.

[8] Chandler Carruth. RFC: Speculative load hardening (a Spectre variant 1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html, 2018.

[9] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[10] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical foundations for software Spectre defenses. In *IEEE Symposium on Security and Privacy (SP)*, 2022.

[11] Rutvik Choudhary, Alan Wang, Zirui Neil Zhao, Adam Morrison, and Christopher W Fletcher. Declassiflow: A static analysis for modeling non-speculative knowledge to relax speculative execution security measures. In *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[12] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. Speculative privacy tracking (SPT): Leaking information from speculative execution without compromising privacy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[13] curve25519-donna. https://code.google.com/archive/p/curve25519-donna/, Accessed 05-04-2024.

[14] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. ProSpeCT: Provably secure speculation for the constant-time policy. In *USENIX Security Symposium*, 2023.

[15] Jacob Fustos, Farzad Farshchi, and Heechul Yun. SpectreGuard: An efficient data-centric defense mechanism against Spectre attacks. In *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[16] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *IEEE Symposium on Security and Privacy (SP)*, 2021.

[17] Ali Hajiabadi, Andreas Diavastos, and Trevor E Carlson. NOREBA: a compiler-informed non-speculative out-of-order commit processor. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[18] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[19] Affected processors: Guidance for security issues on intel processors. https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html, Accessed 20-11-2023.

[20] Speculative execution side channel mitigations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html, Accessed 20-11-2023.

[21] Daniel A Jiménez, Heather L Hanson, and Calvin Lin. Boolean formula-based branch prediction for future technologies. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[22] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A Jiménez, and Baris Kasikci. Whisper: Profile-guided branch misprediction elimination for data center applications. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[23] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation. In *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[25] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael B Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT@ USENIX Security Symposium*, 2018.

[26] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 39–53. IEEE, 2020.

[27] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[28] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. CACTI-P: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel

memory from user space. In *USENIX Security Symposium*, 2018.

[30] Hardware lock elision overview. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/hardware-lock-elision-overview.html, Accessed 23-11-2023.

[31] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient Non-Observability. In *USENIX Security Symposium*, 2021.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM conference on Programming language design and implementation (PLDI)*, 2005.

[33] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[34] Nicholas Mosier, Hamed Nemati, John C Mitchell, and Caroline Trippel. Serberus: Protecting cryptographic code from Spectres at compile-time. In *IEEE Symposium on Security and Privacy (SP)*, 2023.

[35] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Mirage cores: The illusion of many out-of-order cores using in-order hardware. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[36] Arash Pashrashid, Ali Hajiabadi, and Trevor E Carlson. Hidfix: Efficient mitigation of cache-based Spectre attacks through hidden rollbacks. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023.

[37] Popping the hood on golden cove. https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/.

[38] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for websites within the browser. In *USENIX Security Symposium*, 2019.

[39] Eric Rotenberg, Steve Bennett, and James E Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1996.

[40] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1997.

[41] Gururaj Saileshwar and Moinuddin K Qureshi. CleanupSpec: An" undo" approach to safe speculation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[42] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2019.

[43] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A generic approach for mitigating Spectre. In *The Network and Distributed System Security Symposium (NDSS)*, 2020.

[44] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[45] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Robust and scalable process isolation against Spectre in the cloud. In *European Symposium on Research in Computer Security*, 2022.

[46] scikit-bio python library. https://scikit.bio/docs/latest/index.html, Accessed 23-11-2023.

[47] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[48] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre declassified: Reading from the right place at the wrong time. In *IEEE Symposium on Security and Privacy (SP)*, 2023.

[49] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. Typing high-speed cryptography against spectre v1. In *IEEE Symposium on Security and Privacy (SP)*, 2023.

[50] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[51] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.

[52] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[53] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020.

[54] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[55] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with blade. *ACM Symposium on Principles of Programming Languages (POPL)*, 2021.

[56] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[57] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[58] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *The Network and Distributed System Security Symposium (NDSS)*, 2019.

[59] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[60] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N Patt. BranchNet: A convolutional neural network to predict hard-to-predict branches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[61] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *USENIX Security Symposium*, 2023.

[62] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. Speculation invariance (InvarSpec): Faster safe execution through program analysis. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[63] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.