Cassandra: Efficient Enforcement of Sequential Execution for Cryptographic Programs

Ali Hajiabadi*
ETH Zürich
Zürich, Switzerland
ahajiabadi@ethz.ch

Trevor E. Carlson
National University of Singapore
Singapore, Singapore
tcarlson@comp.nus.edu.sg

Abstract

Constant-time programming is a widely deployed approach to harden cryptographic programs against side channel attacks. However, modern processors often violate the underlying assumptions of standard constant-time policies by transiently executing unintended paths of the program. Despite many solutions proposed, addressing control flow misspeculations in an efficient way without losing performance is an open problem.

In this work, we propose Cassandra, a novel hardware/software mechanism to enforce sequential execution for constant-time cryptographic code in a highly efficient manner. 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 mostly static over different runs, and (2) cryptographic programs are loop-intensive and their control flow patterns repeat in a highly compressible way. These insights allow us to perform an upfront 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. Despite providing a strong security guarantee, Cassandra counterintuitively provides an average 1.85% speedup compared to an unsafe baseline processor, mainly due to enforcing near-perfect fetch redirections.

CCS Concepts

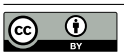
• **Security and privacy → Hardware attacks and countermeasures**; *Cryptography*.

Keywords

Cryptography, constant-time programming, speculative execution, hardware/software co-design

ACM Reference Format:
Ali Hajiabadi and Trevor E. Carlson. 2025. Cassandra: Efficient Enforcement of Sequential Execution for Cryptographic Programs. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731048

*This work was done while the author was at the National University of Singapore.



ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/2025/06
https://doi.org/10.1145/3695053.3731048

1 Introduction

Protecting cryptographic programs has always been a major concern since they are the primary programs that process secrets. While the underlying cryptographic schemes provide strong levels of security to prevent secret extraction through cryptanalysis, their implementations can still be vulnerable to various side channel attacks. *Constant-time programming* is a widely deployed approach to protect cryptographic programs against timing and memory side channels and it is the de facto coding discipline to write high-assurance cryptographic code [2, 7, 54, 86]. Constant-time principles mandate the absence of secret dependent control flow and data flow. In other words, the attacker-visible observations of the execution must be independent of the confidential inputs of the program [3].

Unfortunately, speculative execution of modern processors violates standard constant-time principles that assume instructions are executed sequentially. After the advent of Spectre [35], several speculative execution attacks have demonstrated the ability to leak secrets from verified constant-time programs by transiently declassifying and leaking confidential states [67, 80]. For example, recent attacks have demonstrated powerful adversaries that can manipulate the Branch Prediction Unit (BPU) and precisely control the paths executed by the victim and leak secrets from a constant-time AES implementation [80]. More extensive studies on the constant-time implementation of cryptographic programs demonstrate that most popular libraries leak secrets under speculative execution semantics [4]. Hence, it is essential to find a solution that fundamentally eliminates the speculative execution attack surface in cryptographic programs.

Existing defenses to protect constant-time programs, both on the hardware level [15, 18, 24, 42, 62] and the software level [5, 11, 17, 21, 23, 45, 47, 52, 56, 73–75, 78], deploy a restrictive approach to prevent or limit speculative execution of instructions, diminishing the benefits of speculative, high-performance processors.

While naively disabling data flow speculation shows negligible performance impact for cryptographic programs, addressing control flow speculation is still a major issue for high-performance processors. In this work, we investigate a new, radical design point to strictly enforce sequential execution for cryptographic programs, namely *recording-and-replaying*. This mechanism disables branch prediction altogether, and instead, redirects fetch based on the upfront recorded sequential control flow traces. This design ensures that instruction fetch is always redirected according to the sequential execution model of the program, as assumed by standard constant-time policies. 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 storing/loading these traces in the processor would incur high overheads. In the worst case, it can show a similar slowdown as a processor without a branch predictor which stalls fetch until the branch is resolved.

In this work, 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 with respect to 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 be extremely large (up to millions of decisions per static branch in our evaluated programs). 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 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.

Leveraging these insights, we propose Cassandra, a hardware/software mechanism to enforce sequential execution for cryptographic programs and remove the control flow speculation attack surface within these programs. To the best of our knowledge, Cassandra is the first mechanism that takes advantage of 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 inspired by DNA sequencing techniques that detect frequent and unknown patterns of nucleotides in large DNA sequences [44]. The average size of our new compressed traces is just 20 entries in BearSSL, OpenSSL, and post-quantum crypto primitives.

(2) Microarchitecture (§5). We propose a new processor design that (1) communicates compressed branch traces to the processor, and (2) uses branch traces for fetch redirections while avoiding 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.

Additionally, we provide a detailed security analysis and discussion on how to deploy Cassandra in conjunction with other defenses for a comprehensive Spectre mitigation (§6). Cassandra guarantees sequential execution for cryptographic programs that adhere to a constant-time policy and can be easily integrated with other solutions that block Spectre attacks that violate software isolation (i.e., provide secure speculation for a sandboxing policy [22]).

The main contributions of Cassandra are as follows:

- Introducing a novel recording-and-replaying mechanism to strictly enforce sequential execution for constant-time cryptographic programs;
- Performing a detailed branch analysis and trace compression technique, inspired by DNA sequencing methods, that significantly compresses branch traces;
- Proposing an efficient design of Cassandra that communicates branch traces with the hardware and enforces branch directions of a sequential execution model;
- Achieving a 1.85% speedup over an unsafe baseline processor—delivering performance gains instead of slowdowns—while reducing power consumption by 2.73% and incurring only a 1.26% area overhead.

```
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 implementations of cryptographic applications deploy constant-time principles 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 [3]. In other words, constant-time principles assume an adversary can observe the program counter, memory access patterns, and operands of variable-time instructions, and they guarantee all attacker-visible traces of a program are independent from the confidential inputs of the program [3, 12].

Standard 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 attacks have demonstrated the ability to leak secrets from constant-time programs in modern processors that use a *speculative execution* model [4, 67, 68, 80, 81]. For example, Listing 1 shows a constant-time decryption of confidential input `m`. Sequential execution of the code dictates that the secret state is declassified (line 6) only after all decryption rounds are completed, after which any subsequent leak (line 7) is allowed. 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, hence, violate constant-time policies of the program.

2.2 Speculation Primitives

Speculative execution can be triggered through different sources in modern processors, referred to as *speculation primitives*. Speculation primitives can be categorized into control flow and data flow primitives [10, 12].

Control flow speculation. The Branch Prediction Unit (BPU) in modern processors predicts the next PC after 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 general primitives 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 (e.g., Spectre-v1 [35]).

BTB The Branch Target Buffer (BTB) predicts *indirect branches* (e.g., `jmp [reg]`) to determine the target address of next instruction (e.g., Spectre-v2 [35]).

RSB The Return Stack Buffer (RSB) predicts the target address of return instructions. While returns can considered as indirect branches, processors use the RSB to determine return addresses (e.g. Spectre-RSB [36] and RetBleed [77]).

Note, that commodity microarchitectures might have multiple components that speculate on a specific type of branch. For example, GadgetSpinner [13] demonstrates that the Loop Stream Detector (LSD) in Intel CPUs also speculates on loop conditional branches. However, we use the aforementioned primitives to represent general classes of primitives (i.e., the LSD speculation falls into the PHT 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. Prior attacks demonstrated two primitives that can leak:

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 (e.g., Spectre-v4 [29]).

PSF Predictive store forwarding (PSF) allows a younger load to forward data from an unresolved store before the load and store addresses are resolved (e.g., Spectre-PSF [11]).

Mitigating control flow speculation poses higher overheads compared to data flow. Our experiments in §7.2 show that naively addressing data flow speculation in cryptograhic programs incurs negligible performance overhead (less than 1%). Hence, we only focus on addressing control flow speculation in an efficient way.

2.3 Evolution of Hardware Defenses for Spectre

Early defenses for speculative execution attacks focus only on data caches as the transmission channel, similar to the original Spectre-v1 [34, 50, 51, 60, 61, 79]. More comprehensive defenses, like STT [82] and NDA [76], propose mechanisms to prevent leaks from a more comprehensive list of transmission channels. These solutions implement dynamic taint tracking to restrict the execution or data propagation for instructions that are tainted by speculatively loaded data. While this approach protects *sandboxed* programs [22], 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 [15, 18, 24, 42, 62]. Most hardware-only defenses for constant-time programs introduce additional slowdown compared to the sandboxed cases. This is mainly because they must protect not only speculatively loaded data but also all values that are already loaded in the registers, as any of them can potentially be secret. In this paper, our goal is to strictly enforce sequential execution for cryptographic code and avoid the additional overhead of prior solutions. To the best of our knowledge, our approach is the first that exploits the key characteristics of cryptographic code to improve performance compared to an unprotected baseline, while providing a sequential security guarantee.

Motivating example: DOLMA [42] shows that protecting non-cryptographic programs under a sandboxing policy incurs a 10.2% performance overhead, rising to 22.3% across *all* applications when extended to a constant-time policy; this trend holds for all hardware-based defenses. Cassandra, however, efficiently protects constant-time programs and allows the CPU to select more efficient defenses for other applications which better fit their threat model.

3 Threat Model

Cassandra eliminates the possibility of transient execution exclusively for cryptographic code that adheres to the sequential constant-time policy. Cassandra does not provide protection for software isolation (i.e., sandboxing policy [22]). Existing lightweight isolation techniques [28, 57, 64] or secure speculation mechanisms for sandboxing [24, 42, 76, 82] can be integrated with Cassandra to prevent transient leaks of non-crypto code as well.

We consider Meltdown-type attacks [9, 40, 63, 71, 72] out of scope. These attacks exploit the transient execution upon exceptions and CPU faults, which are efficiently mitigated in recent CPUs via microcode updates [30]. Additionally, non-speculative control flow attacks [19, 25, 55] are out of scope; constant-time programs are inherently safe against such attacks.

4 Branch Analysis of Constant-Time Cryptographic Programs

In this section, we investigate the practicality of a *recording-and-replaying* solution for cryptographic programs to enforce sequential execution. In §4.1, we discuss the key insights that enable our proposed solution, and in §4.2, we detail our branch analysis.

4.1 Key Insights

We discuss two key insights that are directly derived from fundamental characteristics of constant-time cryptographic programs.

> **Insight 1**: *Sequential control flow of constant-time programs is independent of confidential inputs and is determined by the algorithm and its implementation, which are known before execution.*

As we discussed in §2.1, constant-time principles assume that the entire control flow trace and accessed memory addresses are leaked [3]. Hence, the dynamic control flow of the program is required to be independent from confidential inputs. On the other hand, public parameters of the cryptographic programs are specified by standards or determined by the underlying scheme and its implementation, e.g., the key length, array sizes, number of 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.

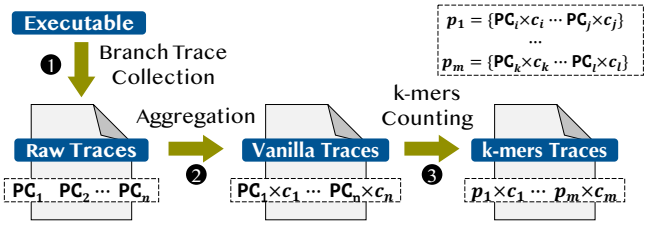


Figure 1: Branch analysis overview in CASSANDRA. Traces are per static branch.

While branch traces of cryptographic programs can be computed before execution, they can still be prohibitively large 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 into a single entry of a small structure in the CPU.

Insight 2: *Sequential control flow of cryptographic programs is highly regular and loop-intensive, allowing for significant compression of control flow traces.*

Most operations and transformations of constant-time cryptographic programs occur in loops (like Listing 1); standard constant-time policies allow 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.

Example: ChaCha20 [46] is a stream cipher that starts with an internal state S as a 4×4 matrix of 32-bit values, consisting of the secret key (256-bit), the nonce (96-bit), a counter (32-bit), and a 128-bit constant, totaling 64 bytes. The encryption of the plaintext P proceeds in four steps: (1) The internal state S is initialized, and state K is initialized with a copy of S. (2) State K is transformed 10 times using the *DoubleRound* function (two rounds of additions, XORs, and rotations); totaling 20 rounds of transformation. (3) Each element of K is added to the corresponding element of S. (4) The first 64 bytes of plaintext P are XORed with K. These steps are repeated until the entire plaintext is encrypted. Notably, all control flow decisions, such as loop counts, calls, and returns, are static and determined by the algorithm, with all operations wrapped in loops.

4.2 Detailed Branch Analysis

In this section, we investigate branches in different constant-time cryptographic programs from BearSSL [54], OpenSSL [48] and post-quantum crypto (PQC) programs: Kyber [37] and SPHINCS+ [69]. We consider all types of branches: conditional direct branches, unconditional indirect branches, returns, etc. To collect branch traces, we use Intel Pin [43] and record the branch target at each execution of a branch. Figure 1 shows an overview of our branch analysis steps. In 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 (we log the next PC for not-taken cases). Here is an example of *raw* trace of a loop branch BR_0 with 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).

Table 1: Branch analysis of cryptographic programs. *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	35.0	2,312	18,677.0	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	7.9	34	494,420.7	8,000,000.0
■	AES-128	163.2	1,530	7.6	50	43.8	510.0
■	ChaCha20	175.8	752	35.5	561	40.9	250.7
■	Poly1305	45.0	600	14.9	134	8.7	200.0
■	SHA-256	3,350.5	31,736	10.7	70	1,077.6	10,578.7
▲	curve25519	19,375.0	128,700	4.3	18	3,479.2	17,000.0
▲	chacha20	24,500.0	32,000	3.0	3	8,166.7	10,666.7
▲	sha256	440.0	44,316	25.8	803	42.2	5,539.5
●	kyber512	738,074.1	34,620,000	5.3	24	89,705.1	2,304,000.0
●	kyber768	1,177,127.1	69,195,000	5.6	54	143,300.9	4,608,000.0
●	sphincs-shake-128s	3,097,903.5	90,110,880	20.5	348	1,019,536.1	30,036,960.0
●	sphincs-haraka-128s	1,863,707.0	59,244,320	24.5	544	599,537.1	19,748,106.7
●	sphincs-sha2-128s	298,160.1	5,300,746	24.6	389	42,948.7	1,766,834.0
	All	637,425.5	90,110,880	19.9	2,312	163,370.7	30,036,960.0

■ BearSSL ▲ OpenSSL ● Post-Quantum Crypto (PQC)

The next step of the analysis builds the *vanilla* traces that are 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 (this is also known as a run-length encoding). Here is the *vanilla* trace of branch BR_0 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 that the average size of *vanilla* traces per branch is 637,425 in our evaluated programs, and the maximum size is 90,110,880[1]. Communicating these large traces with 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 patterns in a given *vanilla* trace.

QUESTION: How does one detect the repeating 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 [6, 44]. For example, two problems in DNA sequencing that can be useful are finding tandem repeats [6] and k-mers counting [44]. A tandem repeat in a DNA sequence is two or more contiguous copies of a pattern of nucleotides. Finding tandem repeats has many applications, like individual identification and tracing the root of an outbreak. k-mers also refer to a substring of size k of a given DNA sequence. Counting the frequency of k-mers is useful in genome assembly and sequence alignment.

4.2.1 k-mers Counting and Traces. In this work, we deploy the k-mers counting technique for pattern repeat detection (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., the TRF tool [6] for tandem repeat finding) and also they are more configurable. We use scikit-bio Python library [65] in our analysis which allows us to define a custom alphabet for DNA sequences,

[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
   Output: k-mers trace K and pattern set P
1  unused_letters ← alphabet \ unique_letters(seq)
2  current_len ← ∞
3  while len(seq) < current_len do
4      current_len = len(seq)
5      coverage.clear()
6      for k ← 2 to max_k do
7          freqs ← count_kmers(seq, k)
8          foreach kmer ∈ freqs do
9              if fres[kmer] > 1 ∧ Size(kmer) ≤ max_k then
10                 coverage[kmer] ← (k × freqs[kmer])/len(seq)
11             end
12         end
13     end
14     most_frequent_kmer ← max(coverage)
15     frequent_kmers.insert(most_frequent_kmer)
16     letter ← unused_letters.pop()
17     seq.replace_and_merge(most_frequent_kmer, letter)
18 end
19 K ← seq
20 P ← frequent_kmers
```

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 ← detect_static_branches(bin_in) Ⓐ
3  foreach branch ∈ unique_branches do
4      [K1, P1] ← generate_kmers_traces(branch, bin_in, inp1)
5      [K2, P2] ← generate_kmers_traces(branch, bin_in, inp2)
6      is_input_dependent ← diff(K1, K2)
7      if ¬is_input_dependent then
8          traces.insert([branch, K1, P1])
9      end
10 end
11 bin_out ← embed_information(bin_in, traces)
12 Procedure generate_kmers_traces(branch, bin, inp)
13     R ← collect_raw_traces(branch, bin, inp) Ⓑ
14     V ← transform_to_vanilla_traces(R) Ⓒ
15     DNA_seq ← transform_to_DNA(V) Ⓓ
16     [K, P] ← kmers_compression(DNA_seq) Ⓔ
17     return [K, P]
```

while most other 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 reduce the storage requirement as much as possible. However, note that we use the *k*-mers counting just as a demonstration and our compression results do not depend on a specific tool.

Before the *k*-mers counting step of our analysis, 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 input of the algorithm is the equivalent DNA sequence of a *vanilla* trace. The core of the algorithm is the *count_kmers* procedure (line 7) that takes k and DNA sequence *seq* as input and builds a frequency map of all the 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 the length of the compressed sequence stops reducing (line 3). 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 the *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). The average *k-mers* size per static branch is 19.9 and the maximum size is 2,312. Compared to *vanilla* trace sizes, our compression leads to an average compression rate of 163,371× and a maximum rate of 30,036,960×. 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 that encrypts data in three encryption rounds with key and plaintext length of two. In the first step, *raw* traces are collected per static branch (step ❶). For instance, `BR6` is a loop branch with a loop count of two: 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 equivalent DNA sequences (step ❸), we perform our *k*-mers branch compression technique and generate the *k-mers* traces and pattern sets (step ❹).

4.3 Automatic Trace Generation Procedure

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 Ⓑ-Ⓔ 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 with two different inputs to detect branches that their traces change depending on the input. For example, *stream loops* in 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[2]. However, all the other branches are wrapped inside this loop and repeat. Hence, they have valid *k-mers* traces. For branches that their trace depends on the input, we stall the fetch until the branch resolves[3]; this incurs a negligible penalty since they are not frequent and quickly resolve.

[2] In addition, some branches in post-quantum crypto primitives have random traces that change in different runs, e.g., two branches in rejection sampling of Kyber.
[3] In general, if traces are not available for a crypto branch, we redirect fetch only if the branch direction is resolved.

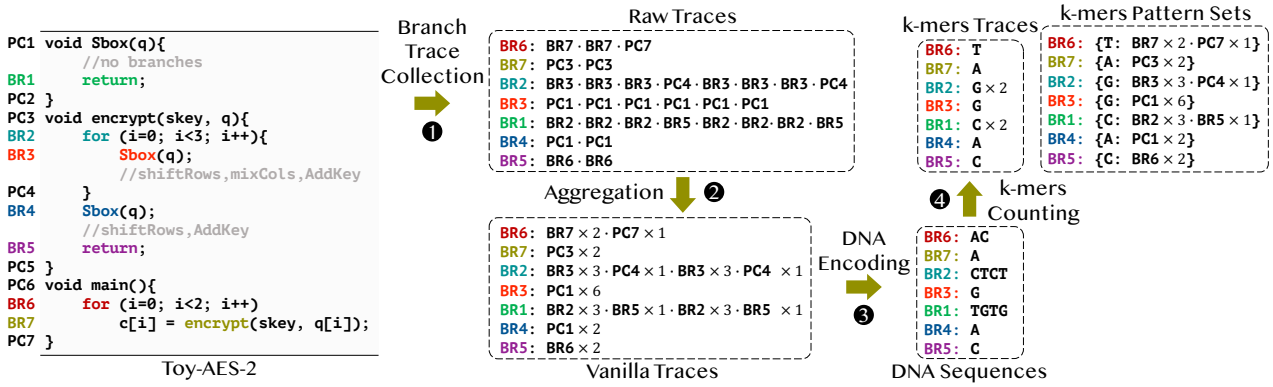


Figure 2: CASSANDRA branch analysis workflow example. Note, that the branches are analyzed separately and traces are generated per static branch; DNA sequences of branches are independent from each other.

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 11, see §5.2 for the details of trace representations and their communication with the hardware). We discuss the runtime overhead of the one-time trace generation procedure for all applications in §7.5.

5 Design of CASSANDRA

To efficiently 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 sequential branch traces. The *BTU* is designed similarly to Trace Caches [58, 59] and Schedule Caches [49] in prior work, with two key differences: (1) traces are determined before execution in CASSANDRA and no dynamic trace selection methodology is used. (2) In case of a trace miss in the *BTU*, the frontend stalls until the trace becomes available, while prior works would switch to a normal, speculative fetch procedure.

In §5.1, we present an overview of CASSANDRA design, and in §5.2 and §5.3, we provide the 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 belongs to a cryptographic program, 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). The *Pattern Table* and the *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 within 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 of the entry (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

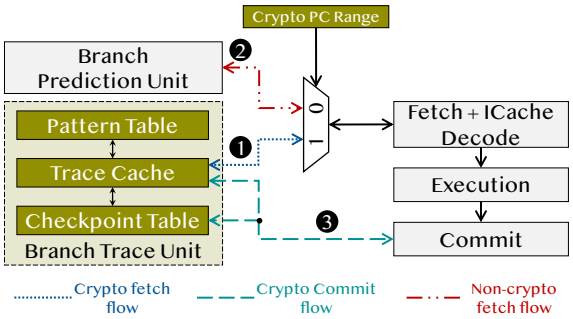


Figure 3: Overview of CASSANDRA microarchitecture. Crypto branches do not access or update the BPU.

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 outcomes, and (2) the branch trace built from the *k-mers* trace K. Figure 4(a) 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 4(b) shows the structure of each element in the branch trace. The first two fields, *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*

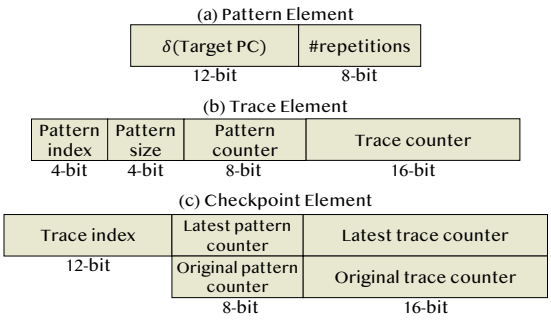


Figure 4: Elements in the *Branch Trace Unit* (*BTU*). Each entry of the *Pattern Table*, *Trace Cache*, and *Checkpoint Table*, consisting of 16 elements and corresponds to a static branch.

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 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 holding 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 avoid additional accesses to bring traces to the *BTU* 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 [33] and RISC-V ISAs [27]. An alternative solution is to re-purpose some of the previously-ignored prefix bytes in x86, in the same way that XRELEASE [41] was implemented, to embed the hint information for each branch (similar to prior work [85]). 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.

Crypto PC range. We also use a new status register that specify PC ranges for crypto code, called the *Crypto PC Ranges* register, to avoid the penalties of waiting until hint information is decoded. Note, that crypto branches that hit in the BTU do not require hint information; only rare cases where traces miss in the BTU require decoding hint information to load traces.

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 4(a));
- *Trace Cache* (*TRC*) holds the branch traces and each entry consists of 16 trace elements (see Figure 4(b));
- *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 4(c)). *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* (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 a *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. Whenever the *pattern counter* reaches zero, we advance to the next pattern element by decrementing the *trace counter* and updating the *pattern counter* based on the new pattern element. 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 and not committed (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 resolve before all elements are looked up.

Non-crypto fetch flow. For non-crypto branches, we use the BPU to determine the next PC (step ❷). However, to prevent speculative fetch redirections to the crypto code we perform an integrity check to prevent fetch redirection if the predicted target is part of the crypto code (using the Crypto PC Ranges register). In this case, we wait for the branch to resolve before taking the branch.

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. To fill the last element of the *TRC* entry, two cases can happen:

(1) if the branch is marked as *short-trace*, a refreshed version of the removed element is inserted at the back of the entry;
(2) if the trace is larger than the *TRC* entry, we prefetch the upcoming elements and insert at the back of the entry. 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 4(c)) 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 evicting 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.

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 or interrupts), 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*.

6 Security Analysis

In this section, we aim to provide a detailed analysis of Cassandra's security and precisely identify its protection scope. We discuss practical deployment and additional considerations to comprehensively block Spectre-type leaks. We explain the preliminaries in §6.1 before discussing Cassandra's security in §6.2.

6.1 Preliminaries

Below, we provide the required background and the terminology used for the security analysis of Cassandra.

6.1.1 Security Policies. Traditionally, developers relied on sequential program execution to enforce security policies in two main application domains: (1) high-assurance cryptography and (2) isolation of untrusted code [12]. We briefly discuss the security policies required in these domains.

Security policy for high-assurance cryptography. As we discussed in §2.1, cryptographic programs deploy *constant-time programming* to ensure that attacker-visible observations (also referred to as *leakage model*) of the program do not depend on secrets. The leakage model of constant-time programs captures the control flow, accessed memory addresses, and operands of variable-time instructions (notated as $[\![\cdot]\!]_{\text{ct}}$ leakage model[4]).

Security policy for software isolation. For software isolation (also referred to as *sandboxing*), a host application needs to ensure that untrusted guest code cannot access the host's memory outside an authorized range (e.g., eBPF in Linux kernel). A common leakage model assumes an adversary observing all architectural computation and accesses, including register file contents ($[\![\cdot]\!]_{\text{arch}}$ leakage model). Thus, this security policy requires preventing out-of-bounds memory accesses during execution.

6.1.2 Spectre Vulnerabilities. Before the advent of Spectre, software-level tools assumed a sequential (architectural) execution model (notated as $[\![\cdot]\!]^{\text{seq}}$ execution model) to enforce either constant-time or isolation policies. However, modern CPUs follow a speculative execution model (notated as $[\![\cdot]\!]^{\text{spec}}$ execution model) that can transiently execute instructions from unintended paths of the program. Figure 5 shows transient leaks of the programs that are secure with respect to a sequential execution model.

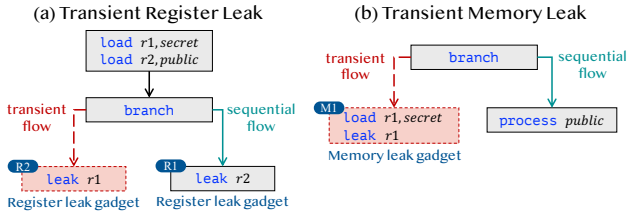


Figure 5: (a) Transient register leak, (b) transient memory leak. Both cases are constant-time during sequential execution, but violated during transient execution.

Sequential execution of both examples in Figure 5 are constant-time (i.e., they are secure according to a $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ contract[5]). Nevertheless, the code in Figure 5(a) presents a transient leak of register $r1$ (through a register leak gadget R2) which contains secret data that never leaks during sequential execution. Similarly, Figure 5(b) transiently accesses a secret memory region and leaks the value loaded in register $r2$ (through memory leak gadget M1).

Moreover, Figure 5(b) demonstrates that the speculative execution model of CPUs can violate software isolation policies as well through transient execution of memory leak gadgets; it leaks a memory location through M1 gadget which is not accessed during sequential execution (i.e., it is secure against a $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ contract).

After the advent of Spectre, many software tools were developed to capture speculative execution semantics and extend constant-time policies to potentially transient program paths [5, 11, 17, 21, 23, 45, 47, 52, 56, 73–75, 78]. However, we argue that software should only consider sequential (non-speculative) semantics, with hardware ensuring no additional leaks beyond those intended by the software. The rationale for this argument is that reasoning about transient execution at the software level requires complete microarchitectural knowledge, which is not public for widely deployed CPUs, making such protections vulnerable to unknown speculation mechanisms. Moreover, software protections lack the performance flexibility of hardware mechanisms, relying on costly measures like fences to block potential transient leaks. Hence, *we present Cassandra as a CPU enhancement that allows developers to write high-assurance cryptographic code using the standard set of constant-time programming rules, without requiring additional programming effort and software-level considerations regarding speculation.*

6.2 Security Analysis of Cassandra

The security goal of Cassandra is to guarantee that (1) all executed paths after crypto branches are on the sequential path, and (2) all crypto leak gadgets execute on the sequential path. In other words, (1) all outgoing edges from BR1 in Figure 6 must follow the sequential (non-speculative) flow of the program, and (2) all incoming edges to M1 and R1 are on the sequential path.

[4] We borrow the notation and terminology from prior work [12, 22].

[5] The combination of a leakage model β and an execution model α expresses a $[\![\cdot]\!]^{\alpha}_{\beta}$ contract that governs the attacker-visible observations of a program during execution.

Table 2: Security analysis of all possible control flow scenarios in CASSANDRA (see Figure 6).

Scenario	Transition ‡	Execution flow	CASSANDRA mechanism
❶	BR1 → R1	branch1 · leak $r1$	Encforcing sequential flow via looking up pre-computed sequential branch traces (*BTU*)
❷	BR1 → M1	branch1 · load $r2, addr_A$ · leak $r2$	Encforcing sequential flow via looking up pre-computed sequential branch traces (*BTU*)
❸	BR1 → R2	branch1 · leak $r4$	Encforcing sequential flow via looking up pre-computed sequential branch traces (*BTU*)
❹	BR1 → M2	branch1 · load $r3, addr_B$ · leak $r3$	Encforcing sequential flow via looking up pre-computed sequential branch traces (*BTU*)
❺	BR2 → M1	branch2 · load $r2, addr_A$ · leak $r2$	Encforcing sequential flow via integrity checks upon non-crypto branches
❻	BR2 → R1	branch2 · leak $r1$	Sequential flow via integrity checks; however, $r1$ is already declassified by the crypto code
❼	BR2 ⋯→ R2	branch2 · leak $r4$	Speculative flow; this is allowed in non-crypto code and $[\![\cdot]\!]^{seq}_{arch}$ contract
❽	BR2 -→ M2	branch2 · load $r3, addr_B$ · leak $r3$	Speculative flow; out of scope (i.e., software isolation)

‡ A: crypto code, B: non-crypto code. $A \rightarrow B$: sequential flow, A -→ B: speculative flow, $A \cdots\!\!\rightarrow B$: don't care flow.

Figure 6 illustrates all possible scenarios that capture the execution of both crypto code (secure against the $[\![\cdot]\!]^{seq}_{ct}$ contract) and non-crypto code (secure against the $[\![\cdot]\!]^{seq}_{arch}$ contract) and their interaction in a CASSANDRA-enabled processor. We will discuss these scenarios to explain CASSANDRA's security and protection scope (crypto gadgets are highlighted as blue and non-crypto gadgets are highlighted as orange). Our security analysis is summarized in Table 2, indicating CASSANDRA mechanisms for each scenario.

Scenarios ❶ and ❷ (BR1 → M1 R1): These are the scenarios where crypto leak gadgets execute after a crypto branch (i.e., leakage of register $r1$ and memory $addr_A$ after branch1). CASSANDRA guarantees sequential execution for these scenarios by looking up the sequential control flow trace of branch1.

Scenario ❸ (BR1 → R2): This is the scenario where a non-crypto register leak gadget executes after a crypto branch (i.e., leakage of register $r4$ after branch1). CASSANDRA guarantees sequential execution for this scenario by looking up the sequential control flow trace of branch1. Note, that the leakage of $r4$ is intentional, as its content should already have been declassified (i.e., made public) before transitioning to unsafe, non-crypto code.

Scenario ❹ (BR1 → M2): This is the scenario where a non-crypto memory leak gadget executes after a crypto branch (i.e., leakage of memory $addr_B$ after branch1). CASSANDRA guarantees sequential execution for this scenario by looking up the sequential trace of branch1.

Scenario ❺ (BR2 → M1): This is the scenario where a crypto memory leak gadget executes after a non-crypto branch (i.e., leakage of memory $addr_A$ after branch2). While a CASSANDRA-enabled processor predicts the outcome of non-crypto branches, we perform an integrity check to not speculatively redirect fetch to crypto code (discussed in §5.3 in the **non-crypto fetch flow**). CASSANDRA guarantees sequential execution for this scenario by stalling fetch until the non-crypto branch resolves.

Scenario ❻ (BR2 → R1): This is the scenario where a crypto register leak gadget executes after a non-crypto branch (i.e., leakage of register $r1$ after branch2). CASSANDRA guarantees sequential execution for this scenario similar to scenario ❺ through integrity checks. Note, that the content of register $r1$ is public since crypto programs declassify registers before transitioning to unsafe, non-crypto code. In other words, transient execution of this scenario would not have leaked any secrets as well.

Scenario ❼ (BR2 ⋯→ R2): This is the scenario where a non-crypto register leak gadget executes after a non-crypto branch (i.e., leakage of register $r4$ after branch2). CASSANDRA allows speculative execution for this scenario. Note, that transient execution of this scenario does not violate software isolation guarantees of the non-crypto code (i.e., it still satisfies the $[\![\cdot]\!]^{seq}_{arch}$ contract).

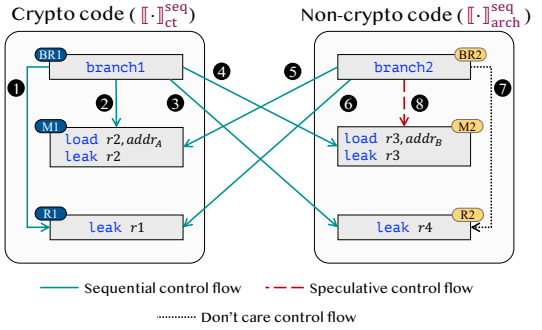


Figure 6: All possible control flows in a CASSANDRA-enabled processor. CASSANDRA guarantees that (1) all outgoing edges from BR1 follow the sequential flow, and (2) all incoming edges to M1 and R1 are on the sequential path.

Scenario ❽ (BR2 -→ M2): This is the scenario where a non-crypto memory leak gadget executes after a non-crypto branch (i.e., leakage of memory $addr_B$ after branch2). CASSANDRA allows speculative execution for this scenario. However, transient execution of this scenario can violate software isolation guarantees of the non-crypto code. Preventing this leakage is out of the scope of CASSANDRA. Since violating software isolation can potentially leak arbitrary memory locations (e.g., secret keys), we expect a CASSANDRA-enabled system to provide a level of isolation for crypto applications (e.g., through lightweight isolation techniques that prevent Spectre [57, 64]). Ideally, CASSANDRA can be integrated with a defense that provides secure speculation for sandboxing policy (e.g., STT [82], DOLMA [42], and Levioso [24]) to comprehensively block Spectre-type attacks for both constant-time and software isolation. Integration of CASSANDRA with other defenses is straightforward; the only consideration is that *crypto branches do not induce a speculation window*, and only speculatively loaded data under the speculation window of non-crypto branches need protection (i.e., scenario ❽).

Formal security analysis. We provide a formalization of CASSANDRA in an extended version of this paper [26], where we demonstrate an interesting use of hardware-software contracts [22]. While hardware-software contracts are mostly used to infer contracts for existing defenses, we provide *contract-informed hardware semantics* with a $[\![\cdot]\!]^{seq}_{ct}$ contract in mind as a clean-slate design (i.e., all fetch

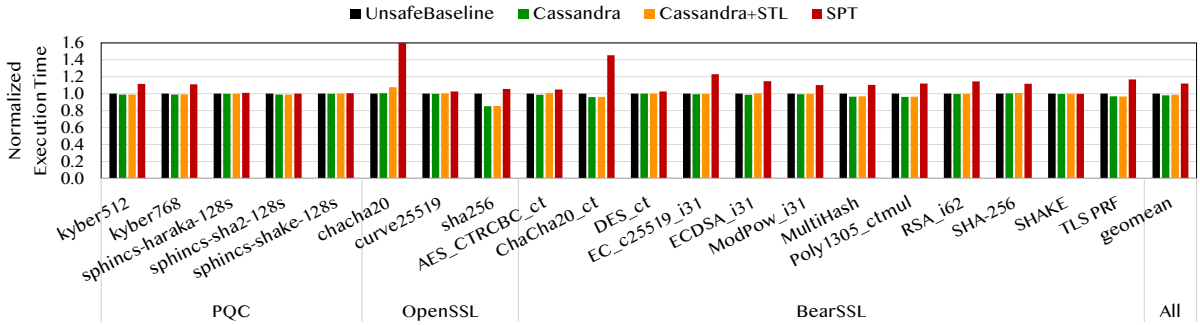


Figure 7: Execution time of different designs normalized to the *Unsafe Baseline*. A higher bar means higher slowdown.

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), LTAGE BPU
BTU	16 *PAT*/*TRC*/*CPT* entries (1.74 KiB storage)
L1 DCache	48 KB, 64 B line, 12-way, 5-cycle latency
L1 ICache	32 KB, 64 B line, 8-way, 5-cycle latency
L2 Cache	1280 KB, 64 B line, 16-way, 14-cycle latency
L3 Cache	30 MB, 64 B line, 16-way, 40-cycle latency

directions strictly follow the contract trace). Hence, the contract satisfaction proof for $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ is a direct result of our contract-informed semantics. We show that our key innovations in trace compression and microarchitecture enable a performant implementation of Cassandra's semantics for cryptographic applications.

7 Evaluation

7.1 Experimental Setup

Simulation. We implement Cassandra on top of the gem5 OoO core implementation and evaluate the design using Syscall Emulation (SE) mode and x86 ISA. Table 3 shows the system configuration (a Golden-Cove-like microarchitecture [53]). We use McPAT 1.3 [38] and CACTI 6.5 [39] to investigate the power and area impacts.

Workloads. We evaluate test applications from the BearSSL [54] and OpenSSL [48] libraries, alongside *reference* implementations of two post-quantum crypto programs: Kyber [37] and SPHINCS+ [69]. For the applications with more than 1B instructions, we used SimPoint [66] to generate representative regions for practical simulation time-frames (an average of 6 SimPoints per application and 50M instructions per region). In §7.3, we evaluate the SpectreGuard synthetic benchmarks [20] as 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 [43] and DynamoRIO [8]).

7.2 Cryptographic Benchmarks Performance

We evaluate four different designs in this section:

- *Unsafe Baseline*: unprotected baseline OoO processor, vulnerable to control flow and data flow speculation;
- Cassandra: our 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 [15, 42];
- *SPT*: a prior hardware-only defense [15]. We use their proposed settings for the Spectre attack model.

Figure 7 shows normalized execution time of the evaluated applications with different designs. Cassandra *improves* performance compared to the *Unsafe Baseline* by 1.85% on average. This is mainly because of the elimination of prediction for crypto branches, and as a result, no ROB squashes and penalties occur due for mispredicting crypto branches.

In addition, the results show that extending Cassandra to protect data flow speculation (i.e., Cassandra+*STL*) achieves a speedup of 1.14%, which demonstrates that Cassandra can still improve performance due to easy-to-resolve address computations in crypto primitives.

Finally, *SPT* shows a 12.07% slowdown compared to the *Unsafe Baseline*, and a 14.21% slowdown compared to the Cassandra. *SPT* has low overheads for some applications, but the overheads can be significantly higher, up to a 59.8% slowdown for OpenSSL `chacha20`, while Cassandra improves performance by 14.7% for OpenSSL `sha256` compared to the *Unsafe Baseline*.

7.3 Synthetic Benchmarks Performance

In this section, we aim to compare Cassandra with *ProSpeCT*, the state-of-the-art secure speculation for constant-time programs.

Implementation. ProSpeCT annotates secret memory regions, and only blocks speculative execution of an instruction if it is about to leak a secret. 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. Also, we declassify all registers at the end of crypto primitives.

Workloads. We evaluate the synthetic benchmark from SpectreGuard [20], 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 crypto primitives: (1) HACL* `chacha20` [86] (similar to

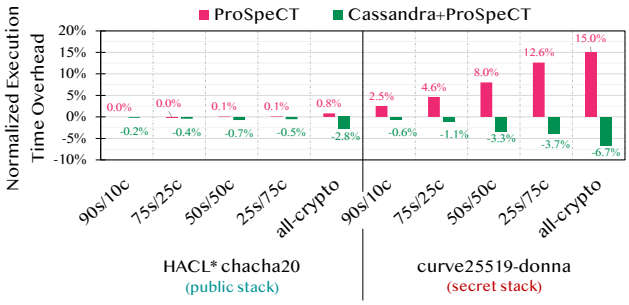


Figure 8: Execution time of *ProSpeCT* and Cassandra, normalized to the respective *Unsafe Baseline* of each benchmark. Negative numbers mean speedup. The stack is marked as public in `chacha20`, and secret in `curve25519`.

ProSpeCT), and (2) `curve25519`-donna [16][6]. The main difference between these two crypto primitives is that `chacha20` does not spill secret variables to the stack, while `curve25519` spills both secret and public variables to the stack which means we need to label the stack as a secret memory region[7].

We evaluate two designs: (1) *ProSpeCT* [18], and (2) Cassandra+*ProSpeCT*. Since Cassandra only enforces sequential execution for crypto branches, we still leave *ProSpeCT* enabled alongside Cassandra to prevent transient memory leaks of annotated secret regions during the non-crypto component (i.e., senario ❽ in Figure 6). Although *ProSpeCT* is not specifically designed as a general solution for software isolation since non-crypto applications do not have a clear notion of secret annotation similar to crypto applications, and all architecturally out-of-bounds memory accesses are confidential. Figure 8 shows the performance impacts of *ProSpeCT* and Cassandra+*ProSpeCT*.

Both *ProSpeCT* and Cassandra have marginal impact on the performance for all benchmark configurations of `chacha20` (Cassandra shows 2.8% improvement and *ProSpeCT* has only 0.8% slowdown for the `all-crypto` case). However, *ProSpeCT* experiences a significant slowdown for `curve25519`; *ProSpeCT* incurs a slowdown between 2.5% and 15.0% when increasing the crypto fraction of the workload. Interestingly, Cassandra provides more speedup when increasing the crypto fraction of the workload, from 0.6% to 6.7%. The main reasons for Cassandra's improvements are: (a) Cassandra does not induce any control flow speculation for the crypto component that benefits from always-correct fetch redirections (i.e., no penalties for crypto branch mispredictions and squashing cycles), and (b) it does not restrict the execution for the crypto component, unlike *ProSpeCT* that needs to label the stack as secret for more complex primitives like `curve25519` and restrict speculative execution of the instructions that potentially process secrets.

Summary. Our experiments demonstrate that the worst-case for Cassandra is marginal/no performance improvement while *ProSpeCT* can experience significant slowdown for complex crypto applications. In addition, *ProSpeCT*'s performance relies on (i) manual programming efforts to precisely annotate secret memory regions, and (ii) how variables are handled with respect to stack spills

[6]We use the secret annotations provided in https://github.com/proteus-core/prospect/
[7]Note that we use a different setup and compiler compared to *ProSpeCT* which impacts the stack spills; we use the Clang v14.0.4 compiler for an x86 target, while *ProSpeCT* uses riscv-gnu-toolchain for a RISC-V target.

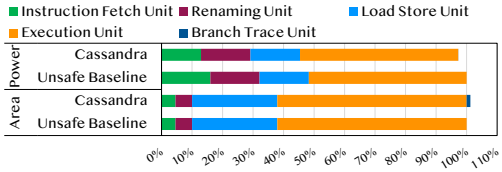


Figure 9: Power and area of Cassandra, normalized to the total power and area of the *Unsafe Baseline*.

which is not trivial to manually isolate secret and public variables. *ProSpeCT* requires a compiler pass to be aware of secret/public annotations when spilling values to the stack to limit the performance overheads. In addition, the compiler might need to co-locate secrets in memory to avoid slowdown due to caching effects [18]. On the other hand, Cassandra does not need any programming effort or new compiler passes to achieve its full performance potential.

7.4 Power and Area Impacts

Figure 9 shows the power and area of Cassandra compared to *Unsafe Baseline*. The results show that Cassandra can reduce the power consumption compared to the *Unsafe Baseline* by 2.73%. The reason is that crypto branches avoid accessing and updating the BPU and 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

We have evaluated the analysis time for each step of the trace generation procedure (steps ❹-❽ in Algorithm 2). We use Intel Pin [43] for dynamic analysis and gathering *raw* traces. Branch detection ❹ is executed once per application and it takes 388 seconds on average. Steps ❺-❽ are executed per branch. The results show that collecting *raw* traces (step ❺) takes 14 seconds on average per branch and *k-mers* compression (step ❽) takes only 3 seconds.

8 Discussion

Q1: Does Cassandra handle branches influenced by public parameters? Constant-time rules mandate all branches to be independent of confidential inputs. However, they can depend on public information. In §4.3, we discussed how to handle branches that their traces change in different runs (e.g., stream loops in stream ciphers). In addition, some branches are influenced by public parameters of the primitive that are specified by standards and underlying algorithms which 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 to generate 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 accessing the proper traces. An alternative solution is to generate separate binaries for each mode.

Q2: Who will provide branch traces and when? Traces need to be re-generated after each compilation only 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.

Q3: What are the benefits and limitations of Cassandra only handling single-target branches? As we discussed in §5.2, many branches in cryptographic programs are *single-target* (they always jump to the same target). Cassandra's branch analysis marks single-target branches and does not consume the *BTU* entries for such branches. A lightweight version of Cassandra, called Cassandra-*lite*, can only support single-target branches and stall fetch for multi-target branches until they resolve (i.e., Cassandra-*lite* does not need to implement a *BTU*). Our evaluations show that Cassandra-*lite* incurs 2.7%, 6.7%, and 4.7% slowdown over Cassandra in BearSSL, OpenSSL, and PQC programs, respectively. Some workloads see significant slowdowns (22% for OpenSSL `sha256`, 8% for `kyber512`) since limiting Cassandra to single-target branches diminishes benefits for conditional branches and returns, where even BPUs struggle. However, Cassandra ensures equal or better performance across complex, widely used applications via near-perfect control-flow (e.g., 14.5% speedup for OpenSSL `sha256` over an *Unsafe Baseline* using the BPU). While Cassandra-*lite* performs better than *SPT*, it lacks Cassandra's performance reliability and improvement and it introduces high overhead for key applications like OpenSSL `sha256` and `kyber512`.

Q4: How are interrupts handled? In Cassandra, the OS does not need to store/reload the *BTU* content during timer interrupts, however, it will flush the BTU if there is a context switch between two different crypto applications (note, that the BTU content is not secret and it is not required to flush it if there is context switch with a non-crypto application). To assess the upper-bound performance impact of flushing the *BTU*, we flush the *BTU* at the frequency of 250Hz [32]; this reduces the performance improvement of Cassandra from 1.85% to only 1.80%.

Q5: Does the *Branch Trace Unit* (*BTU*) introduce a new timing side channel? *BTU* only contains and caches the sequential control flow trace of the program; constant-time principles assume that this trace is already leaked (e.g., ICache has the same leakage), and they guarantee it is completely independent from secrets.

9 Related Work

In this section, we summarize prior work that provide protection for constant-time code against speculative execution attacks. We categorize these solutions into three classes: (1) hardware-only defenses, (2) software-level defenses, and (3) hardware/software co-designed defenses.

(1) Hardware-only defenses. Prior works have investigated hardware defenses to protect constant-time programs [15, 42]. 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 consumes less power compared to the baseline, with modest modifications in the microarchitecture.

(2) Software-level defenses. To harden programs on existing CPUs, compiler passes have been developed that take the speculative execution model of CPUs into account and insert protections as needed [14, 68, 84]. However, software-level defenses usually result in prohibitive slowdowns. Serberus [45] is a compiler mitigation that relies on existing enforcement mechanisms in the CPUs to mitigate all known speculation primitives. Serberus introduces a 21% slowdown on average, and a 8% slowdown on large-buffer benchmarks (8KB buffers). In our evaluation results, a buffer size of 4KB is used in the synthetic benchmark, and the default buffers in BearSSL tests (e.g., `ChaCha20` uses a 400B buffer) [54].

(3) Hardware/software co-designed defenses. Similar to Cassandra, some prior defenses propose hardware/software modifications for efficient Spectre defenses [18, 20, 24, 62, 70, 81]. For example, ProSpeCT [18] manually annotates secrets in the program and blocks speculative execution for instructions that process secrets. We provide a detailed comparison with Cassandra in §7.3.

Profile-guided branch analysis. There have been proposals to use runtime profiles of applications to eliminate branch mispredictions [31, 33, 83]. These techniques mainly target datacenter applications because they have large code footprints and frequent branch mispredictions. For example, Whisper [33] proposes a profile-guided approach that provides hints per static branch to help the branch predictor avoid mispredictions. However, the goal of these works is to build *approximately* accurate branch histories for better prediction, but still rely on speculation.

10 Conclusion

In this work, we propose Cassandra, a novel hardware/software mechanism to enforce sequential execution for constant-time cryptographic programs. To achieve this, we perform upfront branch analysis to significantly compress sequential branch traces which allows an efficient communication with the hardware. During execution, the processor uses the sequential branch traces to determine fetch directions and to avoid accessing the branch predictor. Moreover, Cassandra can be easily integrated with other mechanisms for software isolation and guaranteeing secure speculation for sandboxed programs. Despite providing a strong security guarantee, Cassandra counterintuitively improves performance by 1.85%.

Acknowledgments

We thank the anonymous reviewers for their insightful comments, which contributed to enhancing the final version of this paper. We also thank Arash Pashrashid and Kaveh Razavi for their valuable discussions and feedback.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. In *IEEE International Conference on Data Engineering (ICDE)*.
[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-assurance and high-speed cryptography. In *ACM Conference on Computer and Communications Security (CCS)*.
[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security Symposium*.
[4] Gilles Barthe, Marcel Böhme, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Marco Guarnieri, David Mateos Romero, Peter Schwabe, David

Wu, and Yuval Yarom. 2024. Testing side-channel security of cryptographic implementations against future microarchitectures. In *ACM Conference on Computer and Communications Security (CCS)*.

[5] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-assurance cryptography in the Spectre era. In *IEEE Symposium on Security and Privacy (SP)*.

[6] Gary Benson. 1999. Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Research* (1999).

[7] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*.

[8] Derek Bruening and Saman Amarasinghe. 2004. Efficient, transparent, and comprehensive runtime code manipulation. *Ph.D. Dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering* (2004).

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

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

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

[12] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *IEEE Symposium on Security and Privacy (SP)*.

[13] Yun Chen, Ali Hajiabadi, and Trevor E Carlson. 2024. GadgetSpinner: A new transient execution primitive using the Loop Stream Detector. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[14] Rutvik Choudhary, Alan Wang, Zirui Neil Zhao, Adam Morrison, and Christopher W Fletcher. 2023. Declassiflow: A Static Analysis for Modeling Non-Speculative Knowledge to Relax Speculative Execution Security Measures. In *ACM Conference on Computer and Communications Security (CCS)*.

[15] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. 2021. Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[16] curve25519-donna 2008. curve25519-donna. https://code.google.com/archive/p/curve25519-donna/. Accessed 05-04-2024.

[17] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the haunter-efficient relational symbolic execution for Spectre with haunted relse. In *Network and Distributed Systems Security (NDSS)*.

[18] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. 2023. ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. In *USENIX Security Symposium*.

[19] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A new side-channel attack on directional branch predictor. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

[21] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *IEEE Symposium on Security and Privacy (SP)*.

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

[23] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: Speculative symbolic execution for cache timing leak detection. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.

[24] Ali Hajiabadi, Archit Agarwal, Andreas Diavastos, and Trevor E Carlson. 2024. Levioso: Efficient Compiler-Informed Secure Speculation. In *ACM/IEEE Design Automation Conference (DAC)*.

[25] Ali Hajiabadi and Trevor E Carlson. 2024. Conjuring: Leaking Control Flow via Speculative Fetch Attacks. In *ACM/IEEE Design Automation Conference (DAC)*.

[26] Ali Hajiabadi and Trevor E Carlson. 2025. Cassandra: Efficient Enforcement of Sequential Execution for Cryprographic Programs (Extended Version). *arXiv preprint arXiv:2406.04290* (2025).

[27] Ali Hajiabadi, Andreas Diavastos, and Trevor E Carlson. 2021. 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)*.

[28] Mathé Hertogh, Manuel Wiesinger, Sebastian Österlund, Marius Muench, Nadav Amit, Herbert Bos, and Cristiano Giuffrida. 2023. Quarantine: Mitigating Transient Execution Attacks with Physical Domain Isolation. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

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

[30] intel-affected-cpus [n. d.]. 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.

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

[32] KConfig 2024. Linux Timer Interrupt Frequency Configuration. https://github.com/torvalds/linux/blob/5be63fc19fcaa4c236b307420483578a56986a37/kernel/Kconfig.hz. Accessed 20-08-2024.

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

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

[35] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (SP)*.

[36] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[37] Kyber 2020. Kyber - Cryptographic suite for algebraic lattices. https://pq-crystals.org/kyber/index.shtml. Accessed 20-08-2024.

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

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

[40] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.

[41] Lock Elision 2021. 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.

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

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

[44] Guillaume Marçais and Carl Kingsford. 2011. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics* (2011).

[45] Nicholas Mosier, Hamed Nemati, John C Mitchell, and Caroline Trippel. 2024. Serberus: Protecting Cryptographic Code from Spectres at Compile-Time. In *IEEE Symposium on Security and Privacy (SP)*.

[46] Yoav Nir and Adam Langley. 2018. ChaCha20 and Poly1305 for IETF Protocols. https://www.rfc-editor.org/rfc/rfc8439.

[47] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing spectre-type vulnerabilities to the surface. In *USENIX Security Symposium*.

[48] OpenSSL 2024. OpenSSL - TLS/SSL and crypto library v3.2.2. https://github.com/openssl/openssl/tree/openssl-3.2.2. Accessed 20-08-2024.

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

[50] Arash Pashrashid, Ali Hajiabadi, and Trevor E Carlson. 2022. Fast, robust and accurate detection of cache-based spectre attack phases. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

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

[52] Emmanuel Pescosta, Georg Weissenbacher, and Florian Zuleger. 2021. Bounded model checking of speculative non-interference. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.

[53] Popping the Hood on Golden Cove 2021. Popping the Hood on Golden Cove. https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/.

[54] Thomas Pornin. 2018. BearSSL - Constant-Time Crypto Library. https://www.bearssl.org. Accessed 22-11-2023.

[55] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun. 2021. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *USENIX Security Symposium*.

[56] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. 2021. SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets. In *The Network and Distributed System Security Symposium (NDSS)*.

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

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

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

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

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

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

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

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

[65] scikit-bio 2014. scikit-bio Python Library. https://scikit.bio/docs/latest/index.html. Accessed 23-11-2023.

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

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

[68] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *IEEE Symposium on Security and Privacy (SP)*.

[69] SPHINCS+ 2020. SPHINCS+ - Stateless hash-based signatures. https://sphincs.org/. Accessed 20-08-2024.

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

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

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

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

[74] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2020).

[75] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering* (2019).

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

[77] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *USENIX Security Symposium*.

[78] Meng Wu and Chao Wang. 2019. Abstract interpretation under speculative execution. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

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

[80] Hosein Yavarzadeh, Archit Agarwal, Max Christman, Christina Garman, Daniel Genkin, Andrew Kwong, Daniel Moghimi, Deian Stefan, Kazem Taram, and Dean Tullsen. 2024. Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[81] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *The Network and Distributed System Security Symposium (NDSS)*.

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

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

[84] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *USENIX Security Symposium*.

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

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