MICROSAMPLER: A Framework for Microarchitecture-Level Leakage Detection in Constant Time Execution

Moein Ghaniyoun
The Ohio State University
ghaniyoun.1@osu.edu

Kristin Barber
Google
kristinbarber@google.com

Yinqian Zhang
SUSTech
yinqianz@acm.org

Radu Teodorescu
The Ohio State University
teodores@cse.ohio-state.edu

***Abstract*—Constant-time programming is a principal line of defense against timing side-channel attacks. It involves hardening software in such a manner that execution time is uncorrelated to sensitive data values, and is now broadly employed in most cryptography and other security critical kernels. However, constant-time programming relies on necessary assumptions about the underlying microarchitectural implementation, which are frequently incorrect or incomplete, leading to exploits. Consequently, devising methodologies for joint leakage detection in high assurance applications, compiler optimizations and microarchitectural implementations is an increasingly important problem. This paper presents MICROSAMPLER, a dynamic leakage detection framework to identify *secret-dependent* microarchitectural behavior that can lead to side-channel leakage in security critical software. MICROSAMPLER runs the constant-time code to be verified on a cycle-accurate register-transfer level (RTL) simulation of the target system and builds a comprehensive and detailed representation of microarchitectural state captured at cycle granularity. MICROSAMPLER then uses statistical analysis to measure any existing association between microarchitectural state and data values that are identified as sensitive (e.g. encryption keys). We demonstrate the utility of the proposed leakage detection framework through multiple case studies. We show MICROSAMPLER is able to reveal vulnerabilities in constant-time encryption code in diverse cases where the vulnerabilities originate in the algorithm design, compiler optimizations or microarchitectural implementation.**

I. INTRODUCTION

Over the past decade architecture security has received increased scrutiny. This can be largely attributed to advances in offensive security, which has demonstrated practical, software-controlled *microarchitectural attacks* that use measurable processor characteristics to infer data values from execution activity. Recent exploit disclosures [1], [7], [9], [11]–[14], [24], [30], [32]–[34], [44], [50]–[53], [57], [65], [66] continue to demonstrate the many ways in which confidentiality and integrity guarantees granted by language, software and system-level security measures can be breached using this class of attacks.

Fundamental to microarchitectural attacks is data-dependent execution, which can reveal privileged information through a multitude of side channels. This is especially problematic for security critical applications, which handle sensitive data and may employ software-level hardening like constant-time programming in order to thwart data leakage. These strategies rely on necessary assumptions about the underlying microarchitectural implementation, which are frequently incorrect or incomplete, leading to exploits. In general, violations of data-independent execution assumptions can originate at most levels of the computing stack, including algorithm, programming language, compiler or microarchitecture. It is therefore important to also develop principled, practical and scalable solutions for cross-stack leakage detection.

Prior work on leakage detection of constant-time execution generally falls under two main categories: (1) approaches that test the algorithm/software implementation against a set of principles believed to ensure data-oblivious execution [2], [55], and (2) approaches that verify constant-time or data-independent execution of critical hardware components [18], [19], [22], [28]. We make the case that these approaches, while very useful, are insufficient for practical leakage detection in security sensitive applications running on modern, complex microarchitectures. Tools like DATA [2], [55] cannot capture leakage that originates in the microarchitectural implementation unless it directly manifests as timing leakage at testing time. On the other hand, formal verification approaches aimed at ensuring constant-time execution of certain hardware functional units, do not generally scale to large complex microprocessors.

This work seeks to bridge the gap between software and hardware leakage detection. It presents MICROSAMPLER[1], a dynamic leakage detection framework for automatically identifying microarchitectural state that exhibits correlations with secret data and therefore has the potential to violate constant-time assumptions of security critical software. MICROSAMPLER runs the constant-time code under test on a cycle-accurate RTL simulation of the target system and builds a comprehensive and detailed representation of microarchitectural state captured at cycle granularity. Microarchitectural state traces are captured across multiple executions of *security critical regions* (SCR) within an application, and labeled according to the sensitive data values they process (e.g. encryption keys). MICROSAMPLER then uses statistical analysis to measure any association between microarchitectural states and data values that are identified as sensitive. These cases are then automatically flagged for further analysis.

[1]Available at https://github.com/MoeinGhaniyoun/MicroSampler

It is important to note that not all data-dependent execution necessarily leads to the availability of a timing channel. However, an association between an execution feature, such as data or addresses in microarchitectural buffers, functional unit activity, etc. indicate the potential for creating a timing channel. Importantly, the full visibility into the microarchitectural state at cycle granularity allows MICROSAMPLER to identify potential for leakage without it manifesting directly as timing leakage during verification. This ability sets MICROSAMPLER apart from software-level approaches such as [2], [55]. We also highlight that, even though MICROSAMPLER performs the analysis at microarchitectural level, the data-dependent execution it identifies can originate at higher levels of the stack, including compiler optimizations, implementation bugs or algorithmic vulnerabilities. If secret dependent execution is present, it will manifest in correlations between the cycle-level microarchitectural state and secret data values.

We demonstrate the utility of MICROSAMPLER through several case studies that include a range of vulnerabilities originating in different levels of the computing stack. We focus on the domain of applied cryptography, analyzing several crypto primitives. We utilize the open-source RISC-V BOOM processor as our test bed. We find that MICROSAMPLER is able to identify secret-dependent execution in cryptographic primitives (including a previously unreported vulnerability), leakage introduced by compiler optimizations, as well as microarchitectural features and performance optimizations.

Overall this paper makes the following contributions:

- The first framework we are aware of that enables joint verification of constant-time algorithms, compiler output and microarchitectural implementation.
- Uses a principled statistical approach to identify potential correlations between secret data and microarchitectural state.
- Automatically and precisely flags the sources of correlation at microarchitectural level, even if they do not directly manifest as measurable side-channels.
- Unlike formal approaches, MICROSAMPLER runtime scales linearly with the design size and number of simulation cycles.
- MICROSAMPLER leverages RTL simulations to enable full visibility and coverage of microarchitectural state.
- Makes the case that it is important to jointly test the software and microarchitectural implementation in order to produce trustworthy security critical systems.

The rest of this paper is organized as follows: Section II provides background on constant time programming of secure cryptographic algorithms. Section III discusses related work on security verification. Section IV outlines the threat model. Section V presents the MICROSAMPLER design. Section VI details the experimental methodology. Section VII presents the evaluation and a number of case studies highlighting MICROSAMPLER capabilities. Section VIII concludes.

```
1  uint32 modexp(uint32 a, uint32 mod,
2                unsigned char exp[4]) {
3      int i,j;
4      uint32 r = 1;
5      for(i=3;i>=0;i--) {
6          for(j=7;j>=0;j--) {
7              r = ((uint64)r*r) % mod;
8              if(exp[i] & (1<<j))
9                  r = ((uint64)a*r) % mod;
10             }
11         }
12     return r;
13 }
```

Listing 1: Exemplary Square-and-multiply implementation in C.

II. BACKGROUND

A. Modular Exponentiation

A core operation in many asymmetric ciphers (such as RSA decryption) is modular exponentiation, often implemented with the square-and-multiply algorithm. The square-and-multiply algorithm is the most basic but surprisingly efficient method for performing general exponentiation [35]. The idea is to scan the binary representation of the exponent, starting from the most significant bit and moving to the right. In each iteration, for every exponent bit, the current result is squared. If the currently scanned exponent bit has a value of '1', a multiplication by the base x is also performed. The algorithm implementation is shown in Listing 1. Unfortunately, this traditional implementation has a strong control-flow dependency on the secret exponent and is highly susceptible to leakage.

The crux of the problem with such a verbatim implementation can be seen in Listing 1, where depending on the value of the currently scanned exponent bit, a multiplication will be performed or not (line 9). This translates to a clear discrepancy in execution time during iterations, since iterations operating with an exponent bit of value '1' will execute additional high-latency instructions. A capable attacker can measure the execution time of these iterations and easily infer the value of the key bit being scanned in each, recovering the entire secret key [60].

B. Constant-Time Programming

Constant-time programming, or data-oblivious programming, is the practice of hardening software in such a manner that the same operations are always, unconditionally performed. An algorithm is constant-time where any variation in execution time is uncorrelated with sensitive data.

There are three general principles for writing data-oblivious code [6], [31]:

1) No control-flow depending on secret values
2) No memory accesses where the address depends on a secret value
3) No secrets computed with variable-timing arithmetic

There have been several instances [8], [36], [61] in which a gap between necessary assumptions regarding the underlying hardware and it's true implementation has lead to exploits. Listing 2 shows the square-and-multiply algorithm written to be constant-time. In this version, the squaring *and* the

```
uint32 modexp(uint32 a, uint32 mod,
              unsigned char exp[4]) {
    int i,j;
    uint32 r = 1,t;
    for(i=3;i>=0;i--) {
        for(j=7;j>=0;j--) {
            r = ((uint64)r*r) % mod;
            t = ((uint64)a*r) % mod;
            cmov(&r, &t, (exp[i] & (1<<j)) >> j);
        }
    }
    return r;
}
void cmov(uint32 *r, uint32 *a, uint32 b)
{
    uint32 t;
    b = -b;
    t = (*r ^ *a) & b;
    *r ^= t;
}
```

Listing 2: Constant-time square-and-multiply implementation in C with conditional-copy operation.

multiplication are both always performed regardless of the currently scanned exponent bit's value. The two intermediate results are stored in the variables `t` and `r`, respectively.

The crucial task remaining is to assign the correct intermediate result for the next loop iteration without introducing data-dependent behavior. In other words, it should be indistinguishable whether the intermediate result `t` is selected as the final result for a given iteration. This is achieved by the use of a conditional copy operation, where `t` will be copied into the accumulative result variable `r` *only if* the currently scanned bit indeed has a value of '1'. The conditional copy is a branchless arithmetic assignment, using bit-wise combinations to mathematically select the correct result. Importantly, the same arithmetic instructions are executed regardless of which value is ultimately assigned. The arithmetic assignment can be described by the equation:

$$r = cT + (1 - c)R \tag{1}$$

where c is the control bit used in determining the copy and in this case will be the secret exponent bit.

III. Related Work

Prior work has developed several approaches for detecting leakage at both software and hardware levels. We classify these based on whether they are deployed at system design time (pre-silicon) or after manufacturing (post-silicon).

A. Pre-Silicon Verification

1) Information-flow Tracking: Information-flow tracking mechanisms for hardware designs seek to capture the routes of sensitive data, through architectural [47] or gate-level structures. SecVerilog [64] proposes a type system extension to the Verilog HDL, where information flow policies for a design can be specified and checked statically at compile-time. GLIFT [48] and CellIFT [46] propose gate-level and macrocell-level instrumentation to enforce these flows at run-time. These approaches are effective at identifying a broad range of information leakage scenarios. However, they generally require the redesign of the target processors to support IFT, which may not be practical for all applications due to increased complexity.

2) Dynamic Verification: IntroSpectre [21] leverages targeted fuzzing and RTL simulation trace analysis to discover transient execution attacks. SpecDoctor [25] proposes a multi-phased RTL fuzzer to identify transient execution vulnerabilities through differential testing. WhisperFuzz [10] models microarchitectural events as a directed graph and leverages white-box fuzzing to identify data-dependent timing behavior which is triggered in case two programs with same instructions and different data yield different graph traversal paths. Cascade [45] proposes a pre-silicon CPU fuzzer that separately generates data-flow and control-flow instruction streams which are entangled to create long, complex test sequences. TEESec [20] jointly verifies the trusted execution environments and the underlying microarchitecture via enumerating microarchitectural access paths where the TEE data/metadata could travel. These approaches do not address leakage detection in constant-time code. Although IntroSpectre [21] and SpecDoctor [25] leverage RTL simulation traces similar to MicroSampler, their aim is to identify explicit leakage of data (i.e., verbatim secret found in microarchitectural buffers) whereas MicroSampler applies statistical analysis methods to capture any correlation between microarchitectural states/execution and the program secret values.

3) Formal Techniques: Deutschmann et al. [19] formally verify data-oblivious behavior on small in-order cores (e.g. CVA6 [63]) for different types of instructions. The interaction between instructions is not verified. IODINE [22] and XENON [28] use a similar formal approach and are evaluated on small designs (primarily ALUs and FPUs). They also require annotations of the source code. Checkmate [49] utilizes high-level, formal models of the microarchitecture, called happens-before graphs. Checkmate requires a specification of all legal event orderings (uarch events) for each component of this graph to be comprehensive, employing relational model finding to locate violations of this specification with respect to all possible orderings. Pensieve [59] proposes a microarchitectural modeling framework that uses formal properties of speculative non-interference and leverages bounded model checking to automatically find speculative execution vulnerabilities in the modeled designs. Chroniton [4] verifies constant-time execution by performing a symbolic RTL simulation where the secrets are marked as symbolic variables. While symbolic execution provides strong coverage, it is inherently slow.

B. Post-Silicon Verification

1) Dynamic Verification: DATA [55] collects traces of addresses (instruction and data accesses) referenced during program execution. DATA then looks for variations in addresses across traces with different inputs (keys) through standard statistical testing as a means of leakage detection. MicroWalk [56] uses mutual information between sensitive inputs and program state to detect side-channels. CaType [27] uses a refinement type system to track tainted program variables in bit-level representations. Tainted values are then checked to infer

Tool	Target	Approach	Identifies data-dependent execution in:				Scalability (Design/-Code size, state bits, etc.)	Formal Proof
			Algorithm-Compiler (Control Flow/Mem Accesses)	HW Arithmetic Units	Simple Processor Microarchitecture	Complex Processor Microarchitecture		
DATA [55]	SW	Statistical analysis on memory and control flow address traces	✓	✗	✗	✗	Linear Scalability (10K LOC)	✗
Almeida et al. [2]	SW: Constant-time primitives of OpenSSL	Formal Analysis: Reduction-based	✓	✗	✗	✗	2K LOC	✓
IODINE & XENON [22], [28]	HW: Functional units (ALU, FPU), crypto cores, and very simple processors e.g., ScarvRISCV	Formal Analysis: Solver-aided modular verification	✗	✓	✓	✗	2.5K state bits	✓
Deutschmann et al. [18], [19]	HW: FUs, crypto cores, simple CPUs, OoO processors with abstraction	Formal Analysis: Two-safety property checking	✗	✓	✓	✗*	47K state bits*	✓
MicroSampler	Full System: Hardware, Software (down to assembly)	Statistical analysis on microarchitectural traces	✓	✓	✓	✓	Linear Scalablity (700K state bits)	✗

TABLE I: Comparison of leakage detection and verification tools with a focus on constant-time execution. Target indicates the level at which the analysis is performed. *These works reduce the number of state bits by black-boxing processor components (e.g., ROB). The reported size is the black-boxed version which does not include the entire processor implementation.

any secret dependent branch or memory access. CacheD [54] combines trace recording with symbolic analysis, symbolically executing instructions that could be influenced by the secret key to find compromised data-dependent accesses. Abacus [5] extends CacheD to include secret dependent control-flow while also quantifying each leakage channel. SVF [17] proposes a quantification metric to assess cache leakage, based on correlations between architecturally-simulated observations and a leakage model.

Post-silicon approaches do not have direct access to microarchitectural state and can therefore miss leakage that is not directly triggered during testing.

2) Formal Techniques: Almeida et al. [2] verify constant-time behavior by reducing the security of a program P to the assertion-safety of a program Q, where P is constant-time (w.r.t. the chosen leakage model) if and only if Q is assertion-safe. Such approaches verify constant time properties under certain assumptions about microarchitectural behavior that cannot be guaranteed can turn out to be incorrect or incomplete.

C. Contributions Over Prior Work

The aforementioned approaches have made important contributions, but also have some limitations. We summarize some of the work aimed at identifying/verifying constant-time that is closest to MICROSAMPLER in Table I. While software binary instrumentation frameworks (i.e, DATA [55], CaType [27], Abacus [5]) can scale to larger workloads, they miss any data-dependent execution at microarchitecture-level that is not architecturally exposed (e.g., transient execution, data-dependent optimizations in arithmetic units). These tools only consider secret dependent memory accesses and branches. Formal methods [19], [22], on the other hand, cannot be easily scaled to complex out-of-order (OOO) designs and are mostly limited to verifying individual hardware units (e.g., arithmetic units, simple pipelines). Frameworks like IODINE [22] or XENON [28] are evaluated on hardware blocks on the order of 2.5K state bits. Some formal tools such as [18], [19] developed methods of abstraction by "black-boxing" complex microarchitectural components (e.g. ROB, cache) – replacing their implementation with behavioral models – in order to improve the scalability to larger designs (∼50K state bits). This black-boxing can oversimplify the underlying microarchitecture causing the tool to miss complex microarchitectural interactions, such as speculative execution and data-memory dependent prefetcher activities.

Unlike these prior works, MICROSAMPLER is uniquely focused on cross-stack leakage detection in constant-time execution. It scales linearly with both design size and number of verification cycles. This allows MICROSAMPLER to be deployed on the largest version of the RISC V BOOM (with approximately 700K state bits) in reasonable time. Moreover, MICROSAMPLER uses the actual hardware implementation with no modeling assumptions, fits directly into traditional hardware design workflows and requires no change to the architecture in order to enable execution logging.

IV. THREAT MODEL

MICROSAMPLER is designed for pre-silicon security testing and assumes the verification tool has access to the RTL implementation of the target processor and the source code of the application to be verified. It is intended for analysis of high assurance applications such as constant time primitives where security critical code and data can be easily identified, and desired security properties can be enumerated. Many full-stack vendors like NVIDIA, Apple, Intel, etc. develop their own crypto libraries for both performance and security reasons.

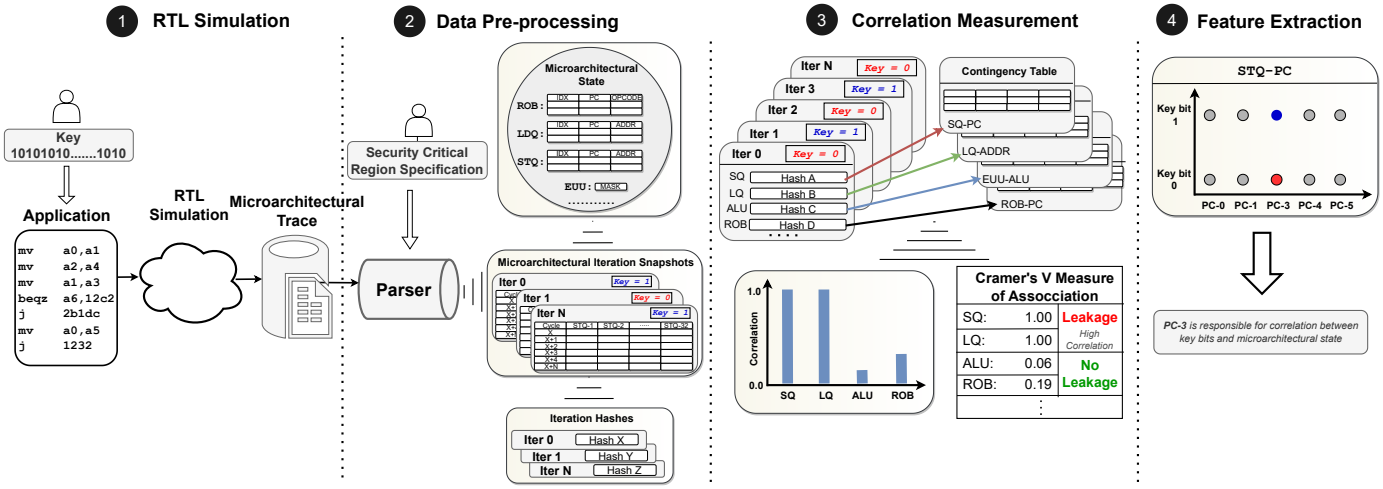


Fig. 1: High-level analysis flow of MICROSAMPLER while verifying microarchitectural state samples from the classic Square-and-Multiply algorithm in Listing 1.

MICROSAMPLER provides the ability to test these crypto libraries on their own hardware and help them identify hard to catch potential vulnerabilities. MICROSAMPLER pinpoints microarchitectural behavior that exhibits statistically significant correlations with secret values.

MICROSAMPLER seeks to identify all cases of secret-dependent microarchitectural state/execution, even if they do not manifest directly as timing leakage. We do not make assumptions about how this state could be leaked. An attacker could induce leakage of such information through a side channel, for example by co-locating with the victim or through other means. We do not assume any special privilege with regard to the attacker capabilities. Our threat model assumes an attacker with the same capabilities as other side-channel attacks such as Flush+Reload [60] or Prime+Probe [37].

V. MICROSAMPLER VERIFICATION FRAMEWORK

The goal of MICROSAMPLER is to identify processor activity that is correlated with sensitive data values computed by constant-time code. To explain our approach, let us consider the square-and-multiply (SAM) Algorithm from Section II-A. For the SAM algorithm, an implementation is considered free of leakage if there exists no statistically significant correlation between microarchitectural state and secret values. The intuition is that if a state appears with high probability during iterations when the secret exponent's value is '1' and low probability when it is '0' (or vice versa) it indicates an opportunity for secret data leakage. At a high-level, MICROSAMPLER is conducting a form of differential analysis for microarchitectural state, accomplished in the following steps and further outlined in Figure 1.

The verification process begins by generating detailed microarchitectural execution traces through RTL simulations of the target processor, while running the code under verification ❶. The state space is then partitioned according to relevant program structure, to capture and tag the microarchitectural state with the sensitive data values processed during execution ❷. Next, statistical analysis is performed to identify correlations between microarchitectural execution and sensitive data values ❸. Finally, if correlation is identified, a feature extraction phase isolates the microarchitectural features that are most responsible for the identified correlation ❹. The results of this analysis can be used to determine if side-channels can potentially exploit the identified vulnerability.

A. RTL Simulation of Target Code

Step ❶ of Figure 1 illustrates that MICROSAMPLER runs the target application on an instrumented RTL simulator to generate a detailed microarchitectural trace, which records the execution state for each pertinent microarchitectural unit, at cycle granularity. For efficiency, the log targets the region of interest for the target application which includes the code that is expected to exhibit constant time execution.

B. Trace Data Pre-Processing

The detailed execution log is processed by the MICROSAMPLER Parser to create *microarchitectural iteration snapshots*. This is illustrated in Step ❷ of Figure 1. Each snapshot reflects the state of the microarchitecture during the execution of an algorithmic "iteration", for example the computation corresponding to a single bit of an encryption key. The iteration snapshots are represented as 2D matrices consisting of the values (or state) of the microarchitectural unit in each cycle of that iteration. Figure 2 shows an example of multiple such state matrices for the Store Queue (SQ-ADDR), which tracks the destination address of store instructions present in the Store Queue during each iteration. The number of rows in each table is equal to the number of simulation cycles for that specific iteration. Each row represents the state of the Store Queue for a single simulation cycle containing all the store addresses in their original order, with empty/invalid entries replaced with 0s. The columns record the microarchitectural features being tracked (e.g. destination addresses in the case of the Store Queue).

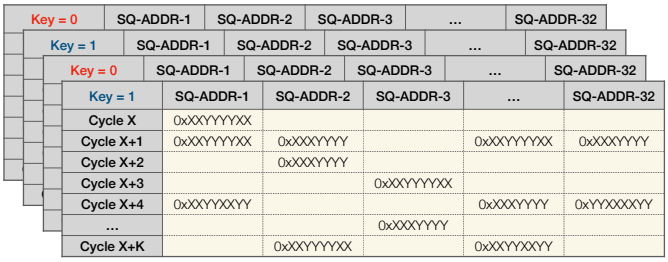


Fig. 2: Microarchitectural iteration snapshots for SQ-ADDR during the processing of different key bits.

In order to facilitate the statistical analysis of the correlation between state snapshots, we generate a unique hash for each distinct microarchitectural state snapshot. A single hash is generated to represent the microarchitectural state for each code iteration that can be mapped to single secret data value (e.g. the execution trace that processes a single key bit). The hashing algorithm ensures that two identical state matrices will generate the same hash, while two distinct matrices have very low probability of producing the same hash (collision). The hashes are 64-bit scalar values generated using Python's default SipHash algorithm.

C. Statistical Correlation Analysis

MICROSAMPLER seeks to measure any statistical association between microarchitectural states and data values that are identified as sensitive.

1) Contingency Tables: In order to perform the statistical analysis, MICROSAMPLER automatically generates contingency tables [40] for each functional unit block, using the frequency of unique hashes representing microarchitectural iteration snapshots corresponding to that block. A contingency table is formed by the multivariate frequency distribution of the variables. These tables are especially useful when exploring the association, dependence, or independence between categorical variables.

To build the contingency table for a microarchitecture unit, we count the frequency of each hash value for each data class (e.g. key bit). Table II shows a contingency table that is formed with the hash-frequency information. Rows represent the output classes (key bit 1 or 0) and columns represent the unique hash values. Values inside the cells represent the count of each hash value for a given class.

Contingency Table for SQ-ADDR					
Hashes / Classes	Hash-0	Hash-1	Hash-2	...	Hash-N
Key = 0	234	131	14	...	43
Key = 1	256	115	10	...	47

TABLE II: A sample contingency table where the number of rows and columns represent the number of output classes and unique hashes, respectively.

2) Quantifying Association Strength: To find the association between nominal values in a contingency table, there exists a few statistical tests that measure the significance of association. We use Cramér's V [16], which measures the association between two nominal variables, producing a value between 0 and 1 (inclusive). It is based on Pearson's chi-squared [38], [39], [41] statistic test. Cramér's V is computed as:

$$V = \sqrt{\frac{\chi^2}{N \cdot \min(k-1, r-1)}} \tag{2}$$

where χ^2 is chi-squared, N is the total number of observations, k is the number of columns and r is the number of rows in the contingency table.

Chi-squared (χ^2) is used to test the independence of categorical variables in a contingency table. It is calculated as:

$$\chi^2 = \sum \frac{(O-E)^2}{E} \tag{3}$$

where O represents the observed frequency and E represents the expected frequency in a cell of the contingency table, calculated under the assumption of independence.

$$E_{i,j} = \frac{\sum O_i \cdot \sum O_j}{N} \tag{4}$$

For each cell (i, j), E is calculated by multiplying the sum of all observations in row i with the sum of all observations in column j, divided by total number of observations N (sum of all cells in the contingency table).

A high Cramér's V value indicates strong correlation between data classes and the microarchitectural state indicating potential leakage for the corresponding functional block. According to [15] correlation is strong for $V > 0.5$. To further validate the statistical significance of the measured correlation, we use Chi-squared's *p-value* that measures the probability of obtaining the same results if the null hypothesis was true. Null hypothesis in MICROSAMPLER denotes the non-existence of correlation between output classes and microarchitectural snapshots. A *p-value* smaller than 0.05 indicates that the measured associativity by Cramér's V is statistically significant.

3) Identifying Correlation Root Causes: Once a functional block is identified to have high correlation, the next step is to help pinpoint which microarchitectural features in that block are most responsible for the observed correlation. Pinpointing the most relevant features (e.g. individual instructions, memory addresses, functional unit activity, etc.) can help MICROSAMPLER users quickly identify the potential vulnerability in the code, compiler optimization and/or microarchitecture.

High correlation manifests as some iteration snapshots (and associated hashes) occurring for some classes and not others. We use two main criteria to identify the microarchitectural features that lead to this correlation. The first is *feature uniqueness*, through which we identify microarchitectural features (e.g. addresses, values, instructions, activity) that are present predominantly in one class but not the other. In order to extract *feature uniqueness*, MICROSAMPLER removes features that are present in all classes, leaving only features unique to each class. This helps pinpoint features such as instructions executed, memory addresses accessed, etc. by only one class.

The second criteria we use for selection is *feature ordering*, which captures event or feature orderings that are unique to each class. For example if the same instructions are present in all classes, but they are scheduled or executed in different order *consistently* across classes they could help pinpoint the source of the correlation. However, because they occur in all classes, they would not be identified by *feature uniqueness*. A chronological ordering of features is instead extracted from the iteration snapshots and compared between classes. Ordering mismatches are reported to the user. Note that this is only done for microarchitectural units for which correlation is observed. MICROSAMPLER will not perform *feature ordering* extraction for units that may exhibit random feature orderings with no correlation.

VI. EXPERIMENTAL SETUP

We evaluate MICROSAMPLER with a system-on-chip design generated by the Chipyard [3] framework. At the center of Chipyard is Chisel, a hardware generation DSL embedded within the Scala language enabling expressive and paramaterizable hardware designs. MICROSAMPLER leverages the *printf* synthesis feature in Chisel, which allows for debugging statements to be carried along through elaboration and accessible from simulator backends. This feature is used to instrument tracing microarchitectural state at cycle granularity, where relevant structures are identified by their high-level Chisel representation and statements transcribing their contents are incorporated into the design with minor modifications. Our prototype SoC used to simulate the case studies proposed in this work is configured to use Chipyard defaults with the `MegaBoom` core; additional core configurations details can be found in Table III.

To run the case study applications, we make use of the `riscv-pk` (proxy kernel) in Chipyard which provides basic system software support by proxying syscalls to the host, bootstrapping the processor, setting up virtual memory and configuring exception handlers. This setup allows for a meaningful testing environment while also curbing runtime overheads. We use Verilator backend simulations from which MICROSAMPLER execution traces are collected.

SoC Configuration	MegaBoom	SmallBoom
No. Cores	1	1
Fetch,Decode,Issue	width=8,4,4	4,1,1
FetchBuffer	entries=32	8
ROB	entries=128	32
PRF	int=128, fp=128	52, 48
LDQ/STQ/LFB	entries=32	8
LFB	entries=64	8
Branch Prediction	type=gshare, entries=2048	gshare, 2048
L1D Cache	sets=64, ways=8, mshr=8, tlb=32	64,4,4,8
L1I Cache	sets=64, ways=8, fetchBytes=16	64,8,8
Prefetcher	type=Next-Line Prefetcher	Same

TABLE III: BOOM Core Configuration.

A. Modular Exponentiation

We select the BearSSL [42] and libgcrypt [23] cryptographic library implementations of the modular exponentiation primitive to use as a baseline. These implementations employ a conditional-copy of intermediate results in order to be constant-time, as described in Section II-B.

B. RTL Simulation

The fields of the microarchitectural structures captured from the features that are input to the statistical analysis phase are listed in Table IV. The selection of which microarchitectural units to include in the execution trace and analysis is done based on design specifications and RTL source analysis. This step can be automated using a compiler pass to identify all sub units. All simulations begin in the same *reset* state.

	Tracked Features	**Feature ID**
Store Queue	Store Address	SQ-ADDR
	Program Counter	SQ-PC
Load Queue	Load Address	LQ-ADDR
	Program Counter	LQ-PC
ROB	ROB Occupancy	ROB-OCPNCY
	Program Counter	ROB-PC
LFB	LFB Content	LFB-Data
	Address	LFB-ADDR
Execution Units	ALU Busy with PC	EUU-ALU
	Address Generator	EUU-ADDRGEN
	Div. Busy with PC	EUU-DIV
	Mult. Busy with PC	EUU-MUL
Prefetchers	Next-line Prefetcher Address	NLP-ADDR
D-Cache	D-Cache Req Address	Cache-ADDR
TLB	TLB Entries	TLB-ADDR
MSHRs	Cache Miss Address	MSHR-ADDR

TABLE IV: Microarchitectural units analyzed by MICROSAMPLER in our case studies.

VII. CASE STUDIES

In this section, we detail multiple case studies selected to illustrate the effectiveness of MICROSAMPLER at identifying subtle cases of data leakage that originate in flawed constant time algorithms, implementations, compiler optimizations as well a microarchitectural features. For each case study we highlight a possible exploit path.

A. Algorithmic/Compiler Vulnerabilities

We showcase three constant-time versions of modular exponentiation. The first case exhibits secret-data dependent behavior introduced by compiler optimizations, the second is rooted in the interplay between algorithm and microarchitecture, and the third version is safe.

1) Case ME-V1-CV*: Constant Time Modular Exponentiation – Compiler Vulnerability:* The *ME-V1-CV* case study tests a constant-time implementation of modular exponentiation. This version is based on the modular exponentiation routine in `libgcrypt` (1.5.3). Listing 3 shows the implementation. The code attempts to camouflage control-flow that is a function of the secret exponent by unconditionally calling the `memmove` function, but assigning the intermediate result to a "dummy" variable in cases where the copy actually should not be performed. At first glance at the C code, it appears that the `ctl` is first checked, and then depending on its value, the `memmove` is called. However, as shown in Figure 3, a high Cramér's V

```
1  void
2  CCOPY_v2(uint32_t ctl, void *dst,
3  void *dummy, const void *src, size_t len)
4  {
5      if (ctl) {
6          memmove(dst, src, len);
7      }
8      else {
9          memmove(dummy, src, len);
10     }
11 }
```

Listing 3: C code implementation of conditional copy in `libgcrypt` used in *ME-V1-CV*.

```
1  BR_CCOPY_v1:
2          mv      a6,ctl
3          mv      a5,a2
4          mv      a0,dst
5          mv      a2,a4
6          mv      a1,a3
7          beqz    a6,2f
8  1:      j       <memmove>
9  2:      mv      a0,dummy
10         j       1b
```

Listing 4: RV64 assembly of *ME-V1-CV* where compiler preloads `dst` before checking `ctl`.

value is observed for almost all microarchitectural units tracked by MICROSAMPLER, indicating a strong correlation between microarchitectural state and the key values.

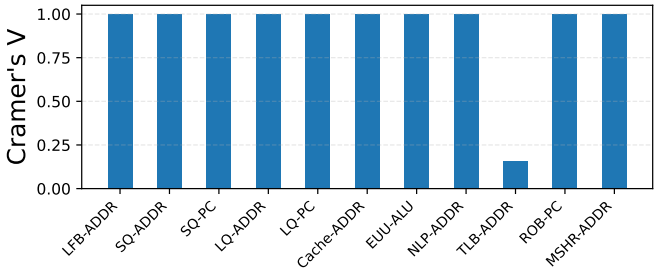


Fig. 3: Cramér's V value for all microarchitectural units tracked while running *ME-V1-CV*.

Upon further analysis, it turns out that compiler optimizations result in an unbalanced assembly code sequence in which the compiler preloads `dst` as the first argument of `memmove` *(a0)* before checking the `ctl` value. As shown in Listing 4, `ctl` value is checked at line 7, if the preloaded value was correct (branch not taken), `memmove` is called. Otherwise, the correct value (`dummy`) is set at line 9 and another jump is executed to line 8. As a result, in cases where `ctl` is 0, two extra instructions (a `mv` and a `jump`) are executed which result in timing discrepancies across key bits.

Possible exploit path: Although `memmove` is eventually executed regardless of key bit value, the extra instructions that are executed only when key bit is 0 can still contribute to a measurable timing difference. An attacker can amplify this timing discrepancy by forcing misprediction on the branch at line 10 Listing 4.

2) Case ME-V1-MV*: Constant Time Modular Exponentiation – Microarchitecture Vulnerability:* We also verify a branchless version of *ME-V1-CV* where bitwise operations are used to check the `ctl` value. Listing 5 shows the implementation of this test case in C.

```
1  void
2  brless_ccopy_v1(uint32_t ctl, void *dst,
3  void *dummy, const void *src, size_t len)
4  {
5      void *dest_dummy = (void *)(dummy ^
6                  (-ctl & (dst ^ dummy)));
7      memmove(dest_dummy, src, len);
8  }
```

Listing 5: Branchless C code of *ME-V1-MV* implementation.

Figure 4 shows the Cramér's V for each functional block while running the *ME-V1-MV* code. We immediately note that, in contrast to *ME-V1-CV*, the measured correlation for half of functional units is lower than 0.2, indicating low secret data correlation for those units. The rest of microarchitectural units with high correlation correspond to memory accesses. Among these is Store Queue Address (SQ-ADDR), which records the destination addresses of store instructions in the Store Queue, each cycle. This indicates that destination addresses of some stores may correlate to key bit values. In order to identify the features most responsible for the high correlation MICROSAMPLER also extracts the unique features for each tracked microarchitectural structure. (Section V-C3).

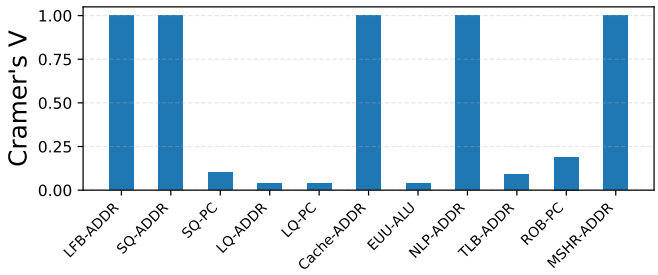


Fig. 4: Cramér's V values for multiple functional blocks while running the *ME-V1-MV* code.

Figure 5 shows the distribution of all features (store addresses) for the SQ-ADDR unit. We observe that some of these features are unique to certain classes. Using the MICROSAMPLER execution log, we automatically identify the instructions that produce these addresses and find that they all belong to the `memmove()` function in the *ME-V1-MV* code (Listing 5). These data-dependent stores to distinct addresses represent a potential vulnerability. We ran the same feature extraction analysis on the rest of the units with high Cramér's V (e.g., LFB-ADDR, NLP-ADDR, etc.) and found that all the identified features are byproducts of store instructions executed by `memmove()`.

Possible exploit path: Secret-dependent store addresses can lead to the formation of a timing channel. For example, flushing one of the memory regions (either `dst` or `dummy`) from the cache will result in an execution time difference depending on the destination of `memmove`, which depends on the value of the secret key.

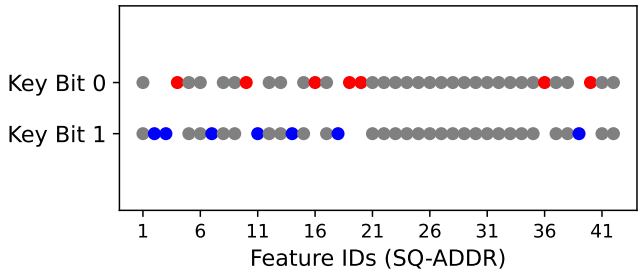


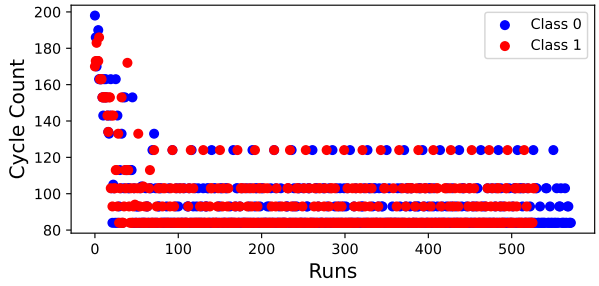
Fig. 5: SQ-ADDR feature uniqueness for *ME-V1-MV*. Red/Blue dots represent unique features (i.e., addresses) in each class.

In order to demonstrate that the identified vulnerability can lead to the formation of a timing channel, we simply initialize one of the memory regions (`dst`) to arbitrary values, which brings them into the cache. This results in cache hits for `memmove` to `dst`. Figure 6b shows the distribution of execution cycle counts across all runs with the `dst` memory region initialized (present in L1D). We can see that the iterations where the initialized value (`dst`) is accessed by `memmove` are almost always slightly faster (lower cycle count) compared to the other iterations. However, Figure 6a (both `dst` and `dummy` are uninitialized) shows largely overlapping distributions making it impossible to distinguish from timing information alone. This shows that MICROSAMPLER is able to detect a potential timing vulnerability even though no timing leakage exists in the *ME-V1-MV* code running under normal conditions. This highlights the importance of joint verification of the software and hardware. MICROSAMPLER is able to automatically pinpoint these potentially vulnerable addresses, first by identifying that the SQ-ADDR is leaky and then by identifying the most correlated features (addresses) that could be exploited in an attack.

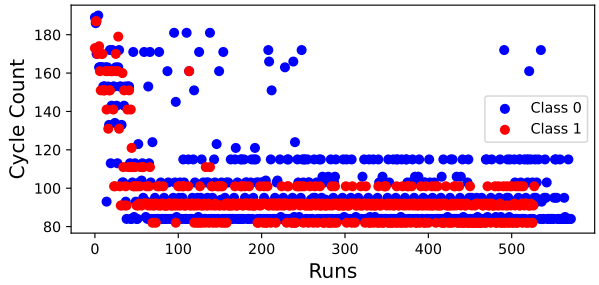
The TLBleed attack [24] found that if the dummy and working set result variables map to different pages, alternating accesses can lead to dTLB misses and a corresponding data-dependent delay, which is measurable through a timing side-channel.

3) Case ME-V2-Safe*: Safe Constant Time Modular Exponentiation:* The *ME-V2-Safe* case study uses the BearSSL library constant-time modular exponentiation. This implementation employs a branchless assignment in the form of Equation 1, with Listing 6 showing the corresponding assembly implementation. The branchless assignment itself is implemented as a boolean expression, trading high-latency arithmetic (which can pose security risks) for logical operations.

Figure 7 shows the Cramér's V measurement across all the microarchitectural units we track, while running *ME-V2-Safe*. The results show that across the board the measured correlation is statistically insignificant. These results reflect the soundness of this constant-time implementation on this particular microarchitecture, with respect to our threat model.



(a) No prior access to either `dst` or `dummy`.



(b) `dst` is initialized prior to sampling.

Fig. 6: Distribution of execution cycle counts across all runs for *ME-V1-MV*.

```
1   CCOPY:
2           add     a3,a3,a2
3           negw    a0,a0
4       1:  bne     a2,a3, 2f
5           ret
6           lbu     a4,0(a1)
7           lbu     a5,0(a2)
8           addi    a2,a2,1
9           addi    a1,a1,1
10          xor     a5,a5,a4
11          and     a5,a5,a0
12          xor     a5,a5,a4
13          sb      a5,-1(a1)
14      2:  j       1b
```

Listing 6: BearSSL conditional-copy implementation used in the `modpow` function. RV64GC.

B. Microarchitectural Vulnerabilities

In this section, we discuss a case study where the cause of data-dependent behavior is rooted in microarchitectural design. We show how complex microarchitectural optimizations such as scheduler optimizations can form data-dependent behavior that is not easy to discover without having full visibility into microarchitectural state.

1) Fast Bypass Implementation in BOOM: Computation simplification methods seek to reduce or eliminate instruction execution when operand values satisfy certain conditions. These techniques have been shown on both complex operations (e.g., square root) and simple integer operations (ADD/AND/OR) [26], [29], [62].

We modify the BOOM processor design to add such an optimization, which we call "fast bypass". Figure 8 shows the implementation of the fast-bypass optimization in the BOOM pipeline (2-wide in this example) and illustrates the effect of this optimization on *ME-V2-Safe* code. In the step ❶,

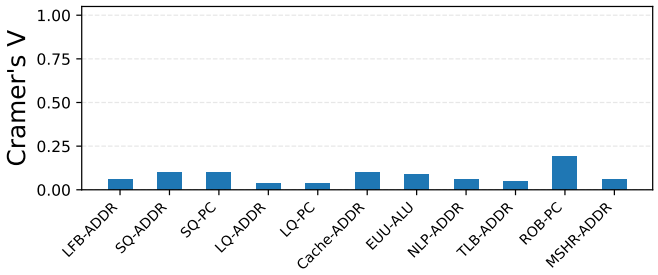


Fig. 7: Cramér's V measurement for all microarchitectural units tracked while running *ME-V2-Safe*.

two instruction (`I1 & I2`) are fetched/decoded/renamed and passed to the issue unit waiting for their operand to become available. ❷ At this stage, two checks are performed on the instructions being renamed: (1) is opcode equal to `AND`, and (2) are the operands of the `AND` available.

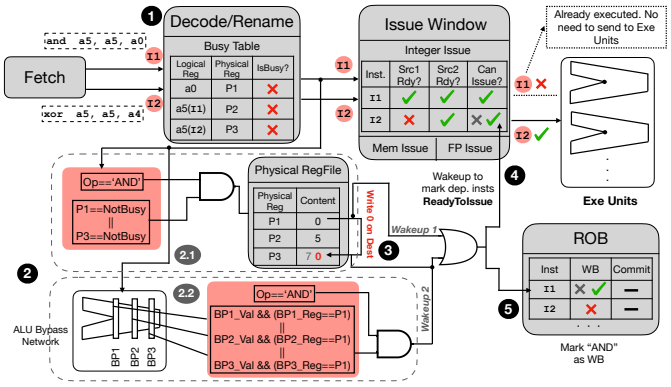


Fig. 8: Fast Bypass optimization on BOOM.

At this point, we have two options: 2.1 If the `AND`'s operands are available, we can get the most updated values from the physical register file. If the `AND`'s operands are still waiting for the completion of a previous instruction 2.2 we check the ALU bypass network. In BOOM, the results of all ALU operations remain in the bypass network for 3 cycles before being written to the register file. The next step ❸ is to check the value of the operand and, if it is equal to zero, we can safely assume this `AND` instruction does not need to be executed and we directly write zero to the `AND` instruction's destination register. Next ❹, we send a signal to the issue window to mark the operands of any instructions dependent on the `AND` as ready. In the case of *ME-V2-Safe*, the `XOR`'s first operand is dependent on the `AND`, and with this optimization in place, it can be issued in the same cycle as the `AND`. In the baseline design the `XOR` had to wait for `AND` to execute and then it could be issued. Finally ❺, a signal is sent to the ROB to mark the `AND` as complete.

2) Case ME-V2-FB *Safe Modular Exponentiation on BOOM Fast Bypass:* We use the MICROSAMPLER framework to verify if the effects of this optimization impact the correct constant time implementation of modular exponentiation from *ME-V2-Safe*. Figure 9 shows the measured Cramér's V values after running the *ME-V2-Safe* code on the BOOM processor with the fast bypass optimization. We can see that, unlike *ME-V2-Safe* which showed no leakage (Figure 7), *ME-V2-FB* shows high Cramér's V for several functional units (Figure 9). When examining the feature uniqueness and feature ordering of the leaking units, we found that no feature in SQ-PC and SQ-ADDR can be attributed to the observed high Cramér's V value. To identify the source of correlation, we removed the timing effects from the SQ-PC and SQ-ADDR iteration snapshots by consolidating consecutive occurrences of the same values to a single value. Next, these snapshots were hashed and the rest of MICROSAMPLER flow is followed. The orange bars in Figure 9 represent the correlation measurement with the timing information removed. We note that the high Cramér's V is no longer present for SQ-PC and SQ-ADDR confirming that the observed correlations for these units were due to timing differences introduced by the fast bypass optimization.

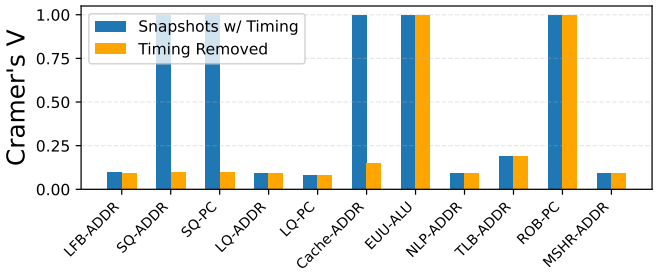


Fig. 9: Cramér's V values for multiple microarchitectural units while running *ME-V2-FB*, for iteration snapshots with and without timing information.

After applying the feature uniqueness test to the ALU, MICROSAMPLER identifies a single instruction that is unique to key bit=1. This is the same `AND` instruction that gets skipped when one of the inputs are '0'. In Listing 2, we can see the control-bit (`b`, representing the key bit value) will be an operand to an `AND` instruction (line 17). This means the fast-bypass optimization is only triggered for key bit values of '0' and consequently the `AND` instruction is only sent to the ALU when the key bit is '1'. Also, using MICROSAMPLER's feature ordering, we identified that the high correlation observed for ROB-PC is due to instruction ordering differences in ROB entries, caused by the fast bypass optimization. When the optimization is triggered, both `XOR` and `AND` instructions occupy the same ROB entry while in normal execution, `XOR` is scheduled first and `AND` goes in the next ROB entry.

This shows that MICROSAMPLER is able to correctly pin-point data dependent execution on the ALU cluster. In the case of *ME-V2-FB*, the fast bypass optimization results in timing leakage that breaks the constant time implementation of the modular exponentiation algorithm. The *ME-V2-FB* case illustrates a previously safe constant time implementations of a security critical primitive that is rendered unsafe due to a simple microarchitectural optimization that can appear benign. This shows the importance of jointly testing security critical code

and the detailed, RTL-level implementation of the architecture.

C. Additional Case Studies

We evaluate MICROSAMPLER on 28 additional constant time primitives from OpenSSL. The full list of all tested primitives is included in Table V. These primitives perform basic operations such as memory comparison, equality checking, conditional swap, etc. and are designed to execute in constant time. MICROSAMPLER analysis reveals no statistically significant correlation between microarchitectural states and the secret inputs for the tested primitives except for the constant-time implementation of memory comparison `CRYPTO_memcmp`.

Constant-time OpenSSL Primitives	**Leakage Identified**
Constant-time Memory Comparison CRYPTO_memcmp()	✓
constant_time_eq/eq_8/eq_int/eq_int_8/eq_bn()	×
constant_time_select/select_8/select_32/select_64()	×
constant_time_ge/ge_s/ge_8_s()	×
constant_time_lt/lt_s/lt_32/lt_64/lt_bn()	×
constant_time_cond_swap/swap_32/swap_64/swap_buff()	×
constant_time_lookup()	×
constant_time_is_zero/zero_s/zero_8/zero_32/zero_64()	×

TABLE V: List of all tested OpenSSL constant-time primitives.

1) Constant Time Memory Compare and Transient Execution: The OpenSSL `CRYPTO_memcmp` constant time memory compare primitive, compares the content of two memory regions in constant time. Listing 7 shows the RISC-V assembly implementation used in several OpenSSL functions. In many of those instances there is an immediate control flow divergence depending on the `CRYPTO_memcmp` return value, as shown in Listing 8. We generated 32 32-byte input values for inputs `a` and `b` with varying distributions of (in)equal bytes selected to increase testing coverage. We extend the MICROSAMPLER region of interest to also sample the microarchitectural traces of a few instructions that use the return value of `CRYPTO_memcmp`.

Figure 10 shows the correlation measurements across all the microarchitectural units we track. We observe low Cramér's V for all microarchitectural traces except the ROB. The MICROSAMPLER feature ordering extraction applied to the ROB helps pinpoint that in some runs, PCs for `equal` and `inequal` are present in ROB in two different stages of program execution. Here is the call patterns we observed:

1) a call to `inequal` then `inequal`.
2) a call to `equal` then `inequal`.
3) a call to `equal` then `equal`.

Naturally, we expect only one call to either `inequal` or `equal` depending on the memory regions pointed to by `a` and `b` in each run. However, these double calls mean that due to misprediction of a branch inside `CRYPTO_memcmp` that checks whether all `len` bytes are compared, the function speculatively returns prematurely and the return value is used to drive the control flow. When the function returns with the correct value (all bytes compared) there is another call to `(in)equal`. By examining the MICROSAMPLER log, we observed that the branch at line 14 in Listing 7 is mispredicted

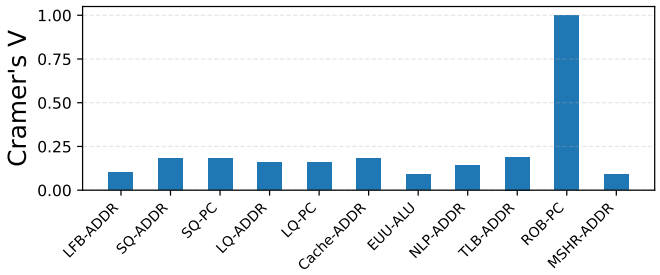


Fig. 10: Cramér's V for all microarchitectural units tracked while running *CT-MEM-CMP*.

```
1  # int CRYPTO_memcmp(const void *in_a,
2  #     const void *in_b, size_t len)
3  CRYPTO_memcmp:
4      li      $x,0
5      beqz    $len,2f    # len == 0
6  1:
7      lbu     $temp1,0($in_a)
8      lbu     $temp2,0($in_b)
9      addi    $in_a,$in_a,1
10     addi    $in_b,$in_b,1
11     addi    $len,$len,-1
12     xor     $temp1,$temp1,$temp2
13     or      $x,$x,$temp1
14     bgtz    $len,1b
15 2:
16     mv      a0,$x
17     ret
```

Listing 7: OpenSSL constant time `CRYPTO_memcmp` RISC-V assembly implementation.

to not taken for the first byte in some runs, resulting in the premature return from `CRYPTO_memcmp`.

Possible exploit path: The partial (speculative) result of `CRYPTO_memcmp` is used to determine the direction of the branch in Line 9 of Listing 8. This creates a secret-dependent control flow divergence that can be exploited by an attacker. We assume a threat model in which the attacker is able to supply inputs to the victim, as well as co-locate an attack process on the same machine as the victim. For example, an attacker could induce a measurable slowdown in execution of one of the secret dependent code paths (e.g. the `equal` function – Line 1 in Listing 8) by flushing its code from the Instruction Cache. The attacker would also mistrain the branch at line 14 in Listing 7 to induce controlled mispredictions. As a result, the victim process will transiently execute either the `equal` or `inequal` paths, depending on the outcome of the byte comparison. Next, the attacker measures the victim's execution time. If the compared bytes were in fact equal, the victim's execution time is slower, due to the iCache miss. By repeating this process with different inputs, the attacker can brute-force the value of the first secret byte. After learning the first byte, the attacker can lock the leaked byte, and re-run the program forcing a branch misprediction after the next byte is compared, repeating the process until the entire secret is revealed.

To the best of our knowledge, this vulnerability has not been observed before. We disclosed this vulnerability to OpenSSL and they reported that local side-channel attacks (i.e., attacker running on the same machine) fall outside of OpenSSL threat

```
uint32_t equal(uint32_t val){
    ...
}
uint32_t inequal(uint32_t val){
    ...
}
uint32_t run(const void *a, const void *b, size_t len){
    volatile uint32_t result;
    if (CRYPTO_memcmp(a, b, len) == 0)
            result = equal(0);
        else
            result = inequal(1);
    return result;
}
```

Listing 8: Control flow dependency on `CRYPTO_memcmp` return value.

model. OpenSSL believe that there is no software fix for this vulnerability.

D. Discussion

Input Coverage: MICROSAMPLER is intended to be used for verifying data-oblivious execution in security critical applications with well-defined secrets (e.g., cryptography algorithms). As many of these algorithms operate on secret values in smaller chunks (e.g., windows of bits), covering all possible valid inputs is feasible. We also utilize a `p-value` test to confirm sufficient input diversity and maintain the statistical significance of observed correlations.

False Positives: MICROSAMPLER could produce false positives in the form of high Cramér's V values when no statistically significant correlation exists. This can be caused by insufficient samples of microarchitectural snapshots (Figure 2). This is more likely to occur if the size of the microarchitectural snapshot is larger (e.g. tracking a larger ROB). We prevent such false positives by measuring the statistical significance of the Cramér's V values using the `p-value` test. We increase the number of inputs to the simulation until the `p-value` falls below a threshold (0.05).

False Negatives: MICROSAMPLER can have false negatives if some secret-correlating structure is excluded from tracking. This can be mitigated by automating the selection of tracked structures. However, MICROSAMPLER is not a formal tool so it cannot prove the absence of vulnerabilities.

Performance and scalability: The MICROSAMPLER performance overhead scales mostly linearly with design size. Table VI lists the performance overhead breakdown for different MICROSAMPLER stages while running *ME-V1-CV* with 4 different 1024-bit keys on `MegaBoom`. We also measured the analysis time for the same test running on the smaller `SmallBoom` configuration (Table III). MICROSAMPLER runs in about 60 minutes with `SmallBoom` and 129 minutes with `MegaBoom`, which is approximately four times larger than `SmallBoom` with respect to size of structures (e.g., ROB). To illustrate MICROSAMPLER's scalability compared to formal verification tools, we include the verification time of XENON [28] for a couple of designs of different sizes: a simple ALU and a small in-order RISC V CPU (SCARV [43]), as reported in [28]. Table VII summarizes the analysis time of MICROSAMPLER for the `SmallBoom` and `MegaBoom` configurations, compared to XENON for the two designs. We can see that the XENON formal tool experiences a 336× increase in analysis time for an 8× larger design. This is in contrast to MICROSAMPLER's analysis time, which only increases by about 2× for a roughly 4× larger design. Note that, while the verification time for SCARV is smaller in absolute terms compared to `SmallBoom` on MICROSAMPLER, it is also a much smaller and simpler design (e.g. in-order vs. out-of-order).

MicroSampler Flow	**Avg. Time Mega-BOOM**
1- Execute program with varying inputs on top of an RTL simulator	∼35mins
2- Extract and parse all traces from the simulation log and generate microarchitectural iteration snapshots	∼51mins
3- Calculate Cramer's V measure of association for all tracked structures	∼30mins
4- Extract feature (PCs, addresses, etc.) responsible for high correlation	∼13mins
Total Analysis Time	∼129mins

TABLE VI: Execution time breakdown for MICROSAMPLER running *ME-V1-CV* with 4, 1024-bit keys (4096 iterations).

	Design (Size)	**Analysis Time**	**Scalability**
MicroSampler	SmallBoom (1x)	60m	4x Size / 2x Time
	MegaBoom (4x)	129m	
XENON [28]	ALU [58] (1x)	2.5s	8x Size / 336x Time
	SCARV [43] (8x)	14min	

TABLE VII: MICROSAMPLER scalability compared to XENON formal verification framework

VIII. CONCLUSION

This paper presented MICROSAMPLER, a cross-stack, scalable security testing framework for leakage detection in constant-time execution kernels. We highlight MICROSAMPLER's ability to reveal vulnerabilities in constant time cryptographic primitives in cases where the vulnerabilities originate in the algorithm, compiler and/or microarchitectural optimizations, including a previously unreported vulnerability in OpenSSL. We make the case that it is of crucial importance to jointly test the software, compiler and microarchitectural implementation in order to produce trustworthy security critical applications and architectures.

ACKNOWLEDGEMENT

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work was also supported in part by the National Science Foundation under awards 2235329 and 2018627.

References

[1] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 870–887.

[2] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying {Constant-Time} implementations," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 53–70.

[3] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[4] A. Athalye, M. F. Kaashoek, N. Zeldovich, and J. Tassarotti, "Leakage models are a leaky abstraction: the case for cycle-level verification of constant-time cryptography," in *Proceedings of the 1st Workshop on Programming Languages and Computer Architecture (PLARCH)*, 2023.

[5] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu, "Abacus: Precise side-channel analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 797–809.

[6] G. Barthe, B. Gregoire, and V. Laporte, "Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time"," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 328–343.

[7] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. V. Rozas, A. Morrison, F. McKeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. R. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes." *arXiv preprint arXiv:2007.11818*, 2020.

[8] D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom, "Sliding right into disaster: left-to-right sliding windows leak," *cryptographic hardware and embedded systems*, vol. 2017, pp. 555–576, 2017.

[9] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[10] P. Borkar, C. Chen, M. Rostami, N. Singh, R. Kande, A.-R. Sadeghi, C. Rebeiro, and J. Rajendran, "Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors," *arXiv preprint arXiv:2402.03704*, 2024.

[11] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[12] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[13] B. Chen, Y. Wang, P. Shome, C. W. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers," in *USENIX Security*, 2024.

[14] Y. Chen, A. Hajiabadi, and T. E. Carlson, "Gadgetspinner: A new transient execution primitive using the loop stream detector," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 15–30.

[15] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[16] H. Cramér, *Mathematical methods of statistics*. Princeton university press, 1999, vol. 43.

[17] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: a metric for measuring information leakage," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, vol. 40, no. 3, 2012, pp. 106–117.

[18] L. Deutschmann, J. Mueller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "A scalable formal verification methodology for data-oblivious hardware," *arXiv preprint arXiv:2308.07757*, 2023.

[19] L. Deutschmann, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "Towards a formally verified hardware root-of-trust for data-oblivious computing," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 727–732.

[20] M. Ghaniyoun, K. Barber, Y. Xiao, Y. Zhang, and R. Teodorescu, "Teesec: Pre-silicon vulnerability discovery for trusted execution environments," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[21] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 874–887.

[22] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, "Iodine: Verifying constant-time execution of hardware," in *Usenix Security*, vol. 19, no. 10.5555, 2019, pp. 3 361 338–3 361 436.

[23] GnuPG, "LIBGCRYPT," https://www.gnupg.org/software/libgcrypt/index.html, 2024, [Online; accessed 4-Dec-2024].

[24] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer : Defeating cache side-channel protections with tlb attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 955–972.

[25] J. Hur, S. Song, S. Kim, and B. Lee, "Specdoctor: Differential fuzz testing to find transient execution vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1473–1487.

[26] M. M. Islam and P. Stenstrom, "Reduction of energy consumption in processors by early detection and bypassing of trivial operations," in *2006 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 2006, pp. 28–34.

[27] K. Jiang, Y. Bao, S. Wang, Z. Liu, and T. Zhang, "Cache refinement type for side-channel detection of cryptographic software," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1583–1597.

[28] R. G. Kici, K. v. Gleissenthall, D. Stefan, and R. Jhala, "Solver-aided constant-time circuit verification," *arXiv preprint arXiv:2104.00461*, 2021.

[29] S. Kim, "Reducing alu and register file energy by dynamic zero detection," in *2007 IEEE International Performance, Computing, and Communications Conference*. IEEE, 2007, pp. 365–371.

[30] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: exploiting speculative execution," *Communications of The ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[31] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*. Springer, 1996, pp. 104–113.

[32] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[33] C. Liu, D. Wang, Y. Lyu, P. Qiu, Y. Jin, Z. Lu, Y. Zhang, and G. Qu, "Uncovering and exploiting amd speculative memory access predictors for fun and profit," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 31–45.

[34] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178*, 2019.

[35] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1996.

[36] A. Moghimi, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations in sgx," in *Cryptographers' Track at the RSA Conference*, 2018, pp. 21–44.

[37] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, pp. 1–20, 2006.

[38] K. Pearson, "Contributions to the mathematical theory of evolution," *Philosophical Transactions of the Royal Society of London. A*, vol. 185, pp. 71–110, 1894.

[39] K. Pearson, "X. contributions to the mathematical theory of evolution.—ii. skew variation in homogeneous material," *Philosophical Transactions of the Royal Society of London.(A.)*, no. 186, pp. 343–414, 1895.

[40] K. Pearson, "Vii. mathematical contributions to the theory of evolution.—iii. regression, heredity, and panmixia," *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character*, no. 187, pp. 253–318, 1896.

[41] K. Pearson, "X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900.

[42] T. Pornin, "BearSSL: a smaller SSL/TLS library," https://bearssl.org/, accessed: 2020-10-07.

[43] SCARV, "SCARV," https://github.com/scarv/scarv-cpu, 2025, [Online; accessed 26-Feb-2025].

[44] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *CCS '19 Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[45] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade:{CPU} fuzzing via intricate program generation," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5341–5358.

[46] F. Solt, B. Gras, and K. Razavi, "{CellIFT}: Leveraging cells for scalable and precise dynamic information flow tracking in {RTL}," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2549–2566.

[47] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, vol. 39, no. 11, 2004, pp. 85–96.

[48] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, vol. 44, no. 3, 2009, pp. 109–120.

[49] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: automated synthesis of hardware exploits and security litmus tests," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 947–960.

[50] P.-A. Tsai, A. Sanchez, C. W. Fletcher, and D. Sanchez, "Safecracker: Leaking secrets through compressed caches," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1125–1140.

[51] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[52] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.

[53] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1491–1505.

[54] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software." in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 235–252.

[55] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "Data - differential address trace analysis: Finding address-based side-channels in binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 603–620.

[56] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference on*, 2018, pp. 161–173.

[57] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, "Inspectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2," in *USENIX Security*, 2024.

[58] XCRYPTO, "XCRYPTO," https://github.com/scarv/xcrypto, 2025, [Online; accessed 26-Feb-2025].

[59] Y. Yang, T. Bourgeat, S. Lau, and M. Yan, "Pensieve: Microarchitectural modeling for security evaluation," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[60] Y. Yarom and K. E. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack," *USENIX Security Symposium*, vol. 2013, pp. 719–732, 2014.

[61] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[62] J. J. Yi and D. J. Lilja, "Improving processor performance by simplifying and bypassing trivial computations," in *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, 2002, pp. 462–465.

[63] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[64] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *Acm Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[65] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans." in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 667–682.

[66] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Last-level cache side-channel attacks are feasible in the modern public cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 582–600.