A Pre-Silicon Approach to Discovering Microarchitectural Vulnerabilities in Security Critical Applications

Kristin Barber, Moein Ghaniyoun, Yinqian Zhang, and Radu Teodorescu

Abstract—Microarchitectural vulnerabilities have become an increasingly effective attack vector. This is especially problematic for security critical applications, which handle sensitive data and may employ software-level hardening in order to thwart data leakage. These strategies rely on necessary assumptions about the underlying microarchitectural implementation, which may (and have proven to be) incorrect in some instances, leading to exploits. Consequently, devising early-stage design tools for reasoning about and verifying the correctness of high assurance applications with respect to a given hardware design is an increasingly important problem. This letter presents a principled dynamic testing methodology to reveal and analyze *data-dependent* microarchitectural behavior with the potential to violate assumptions and requirements of security critical software. A differential analysis is performed of the microarchitectural state space explored during register transfer-level (RTL) simulation to reveal internal activity which correlates to sensitive data used in computation. We demonstrate the utility of the proposed methodology through it's ability to identify secret data leakage from selected case studies with known vulnerabilities.

Index Terms—Hardware security, verification

1 Introduction

Over the past decade hardware security has experienced a renaissance. This can be largely attributed to advances in offensive security research for conducting practical, software-controlled *microarchitectural attacks* which use measurable processor characteristics to infer data values from execution activity.

Fundamental to microarchitectural attacks is data-dependent behaviors, which can reveal privileged information through various types of resource usage. In other words, the operational semantics of some components are influenced by the data *values* they encounter during execution. Therefore, an attack prerequisite is to determine sources of data-dependent behavior in the microarchitecture that can be modulated to expose information.

This work proposes a principled dynamic testing methodology to reveal and analyze problematic, data-dependent microarchitectural behavior with the potential to violate assumptions and requirements of security critical software. We build on top of traditional digital design workflows, where software RTL simulations are instrumented to produce detailed traces containing the contents of microarchitectural structures at cycle granularity. These traces of state observations are then used to identify vulnerabilities related to the space and time resource usage of applications.

A *differential analysis* of execution traces is employed to highlight potential attack vectors, where in each trace the only non-fixed parameter is the sensitive application's data of interest. Microarchitectural state samples are compared across executions of a *security critical region* (SCR) within an application, where these samples have been bucketed into sets according to the sensitive data values potentially influencing that execution. Irregularities (biases) across sets can be assessed by performing membership tests, which indicate data-dependent behavior. We focus on the domain of applied cryptography for case studies on which to apply the proposed methodology, as the secret information in these applications can be readily identified and is well understood. We choose both vulnerable and hardened implementations of crypto primitives to study. We utilize the RISC-V BOOM processor as our testbed and record state traces from RTL simulations during execution of the selected applications. We find that our method is able to identify and confirm processor activity which varies as a function of the secret key for the vulnerable software implementations, as well as the absence of such security property violations for a known, robust constant-time implementation. We also show our framework is able to identify a vulnerable hardware-based optimization that could break the constant-time implementation.

- *Kristin Barber, Moein Ghaniyoun, and Radu Teodorescu are with The Ohio State University, Columbus, OH 43210 USA. E-mail: {barberk, teodores}@cse.ohio-state.edu, ghaniyoun.1@osu.edu.*
- *Yinqian Zhang is with the Southern University of Science and Technology, Shenzhen, Guangdong 518055, China. E-mail: zhangyq3@sustech.edu.cn.*

Manuscript received 20 Dec. 2021; accepted 15 Jan. 2022. Date of publication 14 Feb. 2022; date of current version 3 Mar. 2022.
This work was supported in part by Intel Corp. under the Side Channel Academic Program and by the Air Force Research Laboratory under the Assured and Trusted Microelectronics Solutions award FA8650-20-C-1719.
(Corresponding author: Radu Teodorescu.)
Digital Object Identifier no. 10.1109/LCA.2022.3151256

2 Background

An operation that has been notoriously suseptible to leakage of private data is modular exponentiation used in many asymmertic ciphers (RSA decryption), where the exponent is the secret key. Modular exponentiation is often implemented with the square-and-multiply algorithm. The binary representation of the exponent is scanned, 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 is also performed. The crux of the problem is that depending on the value of the currently scanned exponent bit, a multiplication will be performed or not. 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 [1].

The most common defense against these weaknesses is to employ constant-time programming techniques. There are two general rules for writing constant-time code, (1) no control-flow depending on secret values and (2) no memory accesses where the address depends on a secret value. Constant-time programming is conducted based on a set of best practices; however, there have been several instances in which a gap between necessary assumptions regarding the underlying hardware and it's true implementation has lead to exploits. Listing 1 shows the square-and-multiply algorithm written to be constant-time. In this version, the squaring *and* the multiplication are both always performed regardless of the currently scanned exponent bit's value. The two corresponding intermediate results are stored in the variables `t` and `r`, respectively. The conditional copy is a branch-less 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.

Related Work. Despite detection techniques having a rich history, both static [2], [3] and dynamic in nature, previous approaches primarily focus on the cache and use post-silicon methods (either with direct measurements [4] or relying on what is architecturally visible from binary instrumentation frameworks [5]); at which point, much of the visibility into critical information for reasoning about these vulnerabilities is no longer accessible. Unlike prior work, we examine traces produced by a detailed model of the hardware, therefore not relying on modeling assumptions and having stronger

guarantees. This letter provides the first study to demonstrate how pre-silicon simulation, coupled with tracing infrastructure from recent agile hardware design flows, and differential analysis techniques can be used to search for microarchitectural vulnerabilities in hardware designs *more effectively*.

Listing 1. Hardened SAM implementation in C with conditional-copy operation

```
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;
}
```

3 Detection Approach

The goal of our proposed approach is to uncover processor activity that is correlated with sensitive data values used in computation. Our methodology analyzes the microarchitectural state space over instruction sequences considered to be *security critical regions*.

Threat Model. Our solution is designed for pre-silicon security verification and assumes access to the RTL implementation of the processor-under-test. It is intended for analysis of high assurance applications where security critical code and data can be easily identified and desired security properties can be enumerated. Vulnerabilities identified do not presume a side-channel from which resource modulation is measurable, but rather provides insights about potentially problematic value-dependent execution.

State Space Construction. We first construct a state space representation of the execution trace. A state object is defined at cycle granularity having multiple parameters, where each parameter is a microarchitectural structure. To compare states, it is necessary to define an equivalence relation for these parameters. These definitions of equivalence are inherently dependent on the characteristics and organization of each microarchitectural structure. Equivalence relations are shared for similar structures and classified according to whether they are *multi-dimensional*, where state is encapsulated over several entries (Reorder Buffer, Load Queue, Store Queue, Line-Fill Buffer, Physical Register File); *single-entry*, having a single value (Next-Line Hardware Prefetcher) or *events* expressed as a bit-mask to indicate presence of activity in a given cycle (Functional Unit Occupancy).

Analysis Methodology. The central rationale for the following methodology is based on the hypothesis that *the microarchitectural state affected by data should reveal itself as value-dependent differences in the state samples that occur with high probability.* For example, an implementation of the square-and-multiply (SAM) algorithm is considered free of leakage if the execution of each iteration/round results in the same sequence of microarchitectural states. Under this constraint, finding a state which appears with high probability during iterations when the secret exponent's value is '1' and low probability when it is '0' (or vice versa) indicates a potential violation in requirements.

At a high-level, a form of differential analysis for microarchitectural state is conducted, accomplished in the following six steps and further outlined in Fig. 1. This process compares state samples

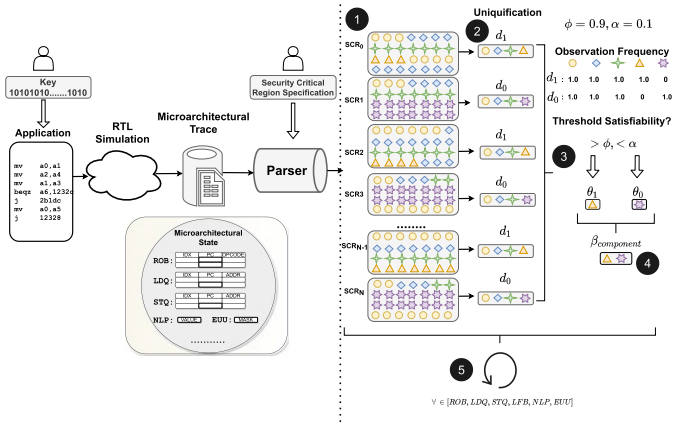

Fig. 1. High-level analysis flow for processing microarchitectural state samples from the classic Square-and-Multiply algorithm.

across executions of the security critical region–during which sensitive application data has distinct values–by ❶ placing samples into sets according to those values. In this way, ❹ irregularities (biases) across sets can be assessed (which are interpreted to indicate data-dependent behavior) by performing membership tests. First, ❷ unique elements are filtered from each set and ❸ *representative* sets are synthesized for each data class by selecting states which appear in a high percentage of iterations with that data value. The number of classes in this example is two ('0' and '1'). Next, a ❹ resultant set, $\beta_{component}$, is created by aggregating these *representative* class sets. This global set holds all microarchitectural state samples constituting statistically significant irregularities across iterations where data values differ. The number of β sets is ❺ equal to the number of microarchitectural structures participating in the analysis and used as a reference for comparison. When any β set is non-zero, this indicates data-dependent behavior. Elements common to $\beta_{component}$ sets across traces ❻ (executions with varying secret keys) provide stronger evidence of correlation.

Similarity Thresholds. Similarity thresholds correspond to the upper and lower bounds used in determining whether a state sample is representative of a particular class and are denoted as ϕ and α, respectively. They provide a means to curate relevant state samples, helping to eliminate false positives.

4 Case Studies

We consider three implementations of modular exponentiation with different degrees of vulnerability to side channel leakage. While our analysis captures most microarchitectural state, we show results for a subset of three important functional blocks: the Load Queue, Store Queue and Reorder Buffer.

Experimental Setup and Applications. We evaluate the approach with a system-on-chip design generated by the Chipyard framework. Our prototype SoC used to simulate the selected case studies is configured to use Chipyard defaults with the `SmallBoom` core. We selected the BearSSL cryptographic library implementation of the modular exponentiation primitive to use as a baseline. Applications are compiled statically for a 64-bit machine with `riscv-unknown-elf-gcc`, optimization level -Os. BearSSL employs a conditional-copy of intermediate results for the modular exponentiation operation in order to be constant-time. We modify this copy operation to create variants which re-introduce known vulnerabilities. Each application variant is run against 50 different 1024-bit keys, where care is taken to select keys with varying distributions of bit values as well as keys with strided bits (regular patterns of alternating values) to increase hysteresis in prediction mechanisms and the probability that they will be triggered. Results are shown for one hundred iterations.

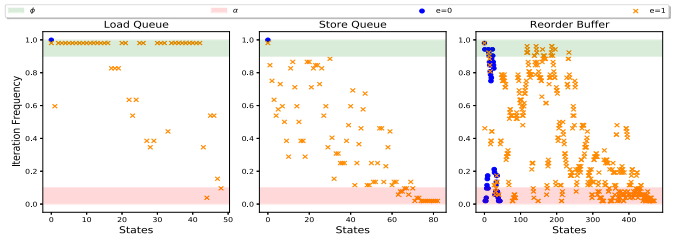

Fig. 2. Microarchitectural state frequency for each data class (d_e) and structure when executing the vulnerable `copy` implementation, with ϕ=.90 and α=.10 highlighted. Each θ_e is composed of the microarchitectural state samples of the corresponding data class (d_e) falling within the region encompassed by ϕ, where the opposite class(es) fall into the α region. Key=`0xaaaa....aaaa`.

Case V1. The *V1* case study uses a canonically vulnerable implementation of modular exponentiation with a strong control-flow dependence on sensitive data. In Fig. 2, the results following step ❸ are shown. Each subplot depicts the frequency of unique states for each distinct data class (d_e) when executing these instructions. The plots show the number of iterations a given state s was observed for d_0 (blue circles) and d_1 (orange crosses) with respect to a particular microarchitectural structure. The green and red regions represent similarity thresholds where ϕ=0.90 and α=0.10. These statistics are used in creating representative class sets for each θ_e. The secret exponent from this trace is strided, in which every other bit flips.

From these plots, clear discrepancies between d_0 and d_1 observations can be seen. Most notably, the LDQ and ROB exhibit several microarchitectural state samples meeting the criteria of the similarity thresholds, with d_1 elements appearing in at least 90% of iterations, while those same elements appear in less than 10% of iterations belonging to d_0. This indicates a strong bias in microarchitectural state content, with respect to the LDQ and ROB, across d_e data classes. This bias manifests through timing leakage, as shown in Fig. 3 with plots of the latency distribution (in cycles) for the execution of the `copy` operation. We can see from Fig. 3a that *V1* has two distinctive regions corresponding to each d_e, separated by roughly 150 cycles.

Case V2. The *V2* case study uses what is intended to be a mitigation for the vulnerability in *V1*, but is an incomplete solution. This version is based on a "fix" incorporated into the modular exponentiation routine in a previous version of `libgcrypt` (1.5.3). Listing 2 shows our implementation introducing the erroneous countermeasure. 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. In other words, the algorithm executes both the square and multiply steps for each bit, but ignores the result of the multiply step for bits of value '0' by effectively discarding it into an unused variable.

In the *V1* application, one could argue the differential signal is amplified since aside from the function preamble and register spilling, executions of the security critical region are entirely disjoint with respect to d_0 and d_1. In the case of *V2*, there is substantially

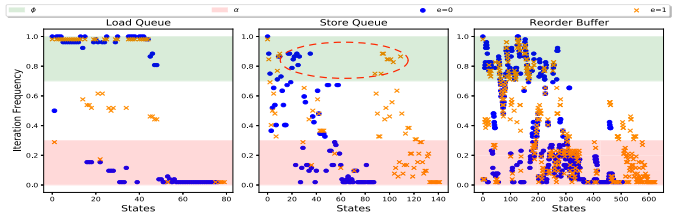

Fig. 4. Microarchitectural state frequency for each data class (d_e) and structure when executing the incorrect mitigation from *V2* in Listing 2, with ϕ=.70 and α=.30.

more overlap in the microarchitectural state space across iterations. This can be seen in the LDQ, STQ and ROB subplots from Fig. 4. An interesting aspect to consider when contrasting *V1* and *V2* is that, at least from a cursory perspective, the *V2* strategy is successful in its objective to reduce the side-channel signal used to exploit the vulnerability from *V1*. As Fig. 3b shows, *V2* (dummy assignment) has largely indistinguishable latency distributions, making it difficult to determine d_e from timing information (iteration latencies) alone.

However, it can be seen from Fig. 4 that in fact there are statistically significant differences across d_e in microarchitectural state (in particular, those inside the dashed red lasso on the Store Queue subplot). This bias established by our approach is corroborated by the findings in the TLBleed attack [6]. TLBleed found if the dummy and working set result variables happen to lie on separate pages, alternating accesses can lead to dTLB misses and a corresponding delay. Indeed, a portion of these biased Store Queue states reference those result variables. This shows that our approach can pinpoint potential leakage through microarchitectural channels even in the absence of a directly-measured timing leakage.

Listing 2. Erroneous countermeasure used in the *V2* implementation

```
void
CCOPY_v2(uint32_t ctl, void *dst,
void *dummy, const void *src,
size_t len)
{
    if (ctl) {
        memmove(dst, src, len);
    }
    else {
        memmove(dummy, src, len);
    }
    return;
}
```

Case CT. The *CT* case study utilizes the BearSSL library constant-time modular exponentiation routine directly, which uses a branchless assignment implemented as a boolean expression. Fig. 6 shows the distribution of state samples for each d_e from the execution of this application using a key with uniformly distributed d_e. We can immediately see regular patterns of state observations across the data classes (with relatively minimal deviation between them), indicating that the same operations are performed irrespective of the

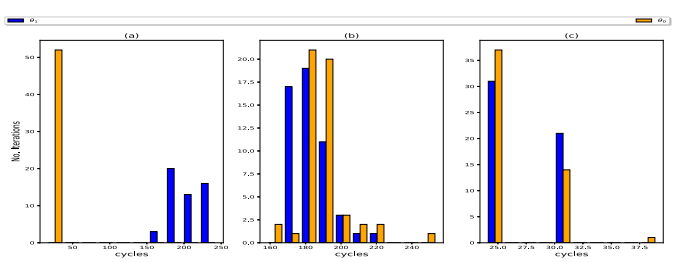

Fig. 3. Latency distribution of `copy` operation. (a) *V1*, control-flow dependency; (b) *V2*, unused result; (c) *CT*, branchless assignment.

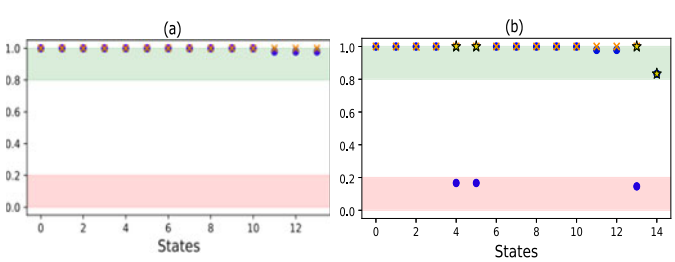

Fig. 5. Execution Unit Utilization of BOOM processor when executing the conditional-copy for (a) baseline and (b) aggressive scheduling implementations.

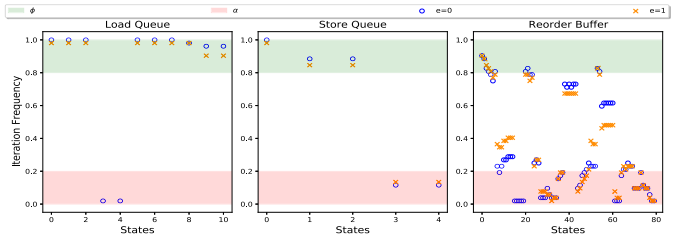

Fig. 6. Microarchitectural state frequency for each data class (d_e) and structure when executing the BearSSL constant-time `copy` implementation, with ϕ=.80 and α=.20.

currently scanned key bit. In fact, we do not find any statistically significant differences in microarchitectural state between the data classes which could reasonably be used to distinguish them (for the microarchitectural structures we analyze). These results confirm the soundness of this constant-time implementation on this particular microarchitecture, with respect to our threat model.

Case CT-V. The following case study demonstrates how this methodology can be used to reveal microarchitectural design choices capable of forming new vulnerabilities. We modify the existing BOOM processor with an optimization to the scheduling logic of the execution cluster. The optimization skips the execution of `AND` and `MUL` instructions if one of their input operands is '0', and forwards their result directly to dependents. An early snoop of the operand values is performed for such instructions during the rename stage. This type of performance optimization can introduce a potential vulnerability to constant-time code. In Listing 1, 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 optimization will only affect key bit values of '0'.

We use our framework to verify if this is observable in the microarchitectural state. Fig. 5 shows the execution unit utilization (EUU) for the baseline BOOM processor (a) and for the version with the scheduling optimization (b). In the optimized version we observe multiple states with distinct correlation to data classes (key values)–these are highlighted as stars in Fig. 5b. The baseline exhibits no such correlation. The reports generated from this analysis show details of instruction residency in execution units for the identified states and confirm that these variations are due to different co-scheduling that results from the eager execution enabled by the scheduling optimization.

5 Future Work and Conclusion

This letter proposed the first treatment of the microarchitectural state space explored during RTL simulation as a means to detect hardware functionality having leakage potential with respect to an application. We present results for a set of implementations of modular exponentiation, demonstrating its utility through an ability to correctly capture known vulnerabilities and *highlighting meaningful information which would be opaque to other methods*.

We expect this technique to extend well in capturing other forms of implicit and explicit value-dependent behavior in the microarchitecture of complex processors, as an aid for reasoning about the security implications of various hardware optimizations on high assurance applications. The statistics reported can also serve as a building block towards more automated forms of detection.

Acknowledgments

The authors would like to thank the members of the Architecture Research Lab at Ohio State, Matthew Fernandez and Yuan Xiao from Intel, and Andres Marquez and Kevin Barker from the Pacific Northwest National Laboratory for valuable feedback on this work.

References

[1] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 719–732.
[2] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, 2015, Art. no. 4.
[3] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller, "Quantifying the information leak in cache attacks via symbolic execution," in *Proc. 15th ACM-IEEE Int. Conf. Formal Methods Models Syst. Des.*, 2017, pp. 25–35.
[4] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?," in *Proc. Des. Automat. Test Eur. Conf. Exhib.*, 2017, pp. 1701–1706.
[5] 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 *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 603–620.
[6] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 955–972.