HASTE: Software Security Analysis for Timing Attacks on Clear Hardware Assumption

Prabuddha Chakraborty, Jonathan Cruz, Christopher Posada, Sandip Ray, and Swarup Bhunia
Department of ECE, University of Florida, Gainesville, FL 32611, USA

***Abstract*—Information leakage via timing side-channel analysis can compromise embedded systems used in diverse applications that are otherwise secure. Most state-of-the-art timing side-channel detection techniques focus on analyzing the software code while paying little to no attention to the underlying hardware. This limits the ability of such techniques in terms of detection and repair. In this paper, we propose a timing side-channel analysis framework that takes into consideration both the software and the underlying hardware micro-architecture to detect vulnerabilities with high precision. We also propose a set of metrics to quantify the severity of the vulnerabilities. We verify our proposed framework on two different computation subroutines which are widely used in crypto and secure systems.**

***Index Terms*—Hardware-Software Co-Security, Embedded Systems Security, Timing Side-Channels.**

I. Introduction

If the execution time of a process depends on the value of secret data (such as an encryption key), then timing side-channel attacks can potentially infer the secret data value. The attack is carried out by obtaining multiple measurements of the execution time of a process directly from inside the system [1] or via remote interactions [2]. Another class of timing side-channel analysis exploits secret dependent memory access patterns to infer the secret [3].

Security analysis in the software domain attempts to address timing side-channel exploits by constant time expressions, padding, obfuscated execution, etc. [4], [5]. Unfortunately, most existing timing side-channel tools are agnostic of the underlying micro-architecture [6], [5] and only a few frameworks incorporate minimal micro-architecture information by modeling cache and memory accesses [7], [8]. However, increasingly complex micro-architectures make it necessary to include much more hardware information to accurately capture potential leakage.

In this paper, we present a framework, HASTE (**H**ardware-**A**ware **S**oftware **T**iming-attack **E**valuation) for accurate and system-specific (hardware + software) timing vulnerability detection and quantification (see Fig. 1). HASTE transforms the software code into a control flow graph (CFG) and performs a taint propagation to detect all information flow from the declared assets. Execution time of all basic blocks in the CFG, on the given hardware (via simulation on BOOM cores) are captured and mapped to the CFG. We formalize three different sub-categories of timing side-channel vulnerabilities and propose algorithms (Algo. 1, Algo. 2, Algo. 3) to detect and quantify those vulnerabilities given the asset tainted CFG and the real hardware execution time information for each basic block. Quantifying the vulnerabilities allows the system designer to address only those vulnerabilities which appear to be most critical given the system deployment setup.

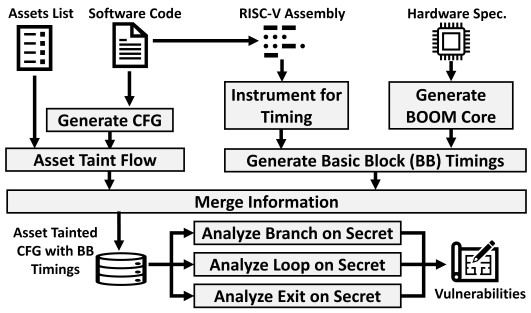

Fig. 1: Overview of HASTE framework: the input-output interface and major steps for automatic vulnerability analysis.

The paper makes the following contributions.

- We create a unique framework for automatic hardware-aware software timing-side channel vulnerability analysis.
- We formalize three different categories of timing side-channel vulnerabilities with well-defined metrics; propose algorithms to detect them; and implement the algorithms into the HASTE framework.
- We demonstrate HASTE by detecting different timing side-channel vulnerabilities for two different secure procedures running on five different micro-architectures.

II. Related Work

Information flow analysis using Program Dependence Graph (PDG) can be used to detect timing side-channel vulnerabilities. Rodrigues *et al.* show how to generate a PDG with $O(|V| + |U|)$ edges, where V is the set of program variables and U is the set of variable uses; this approach is integrated in the FlowTracker [6] tool. Alemeida *et al.* develop another tool, et-verif [5], that detects flaws in constant-time programs using a reduction-based approach. Both FlowTracker and et-verif are hardware agnostic. Recently, Reparraz *et al.* developed a tool, dudect [9], which attempts to identify non-constant time codes by gathering execution time data to detect potential deviations. It requires running the function under test many times to obtain good statistical confidence. In comparison, HASTE can identify specific locations of code that causes the timing vulnerabilities and is more scalable due to minimal dynamic/simulation effort. MicroWalk [7] uses cryptographically secure pseudorandom number generators for creating and applying input vectors for execution trace generation. However, its effectiveness and scalability on other programs have not been explored. CaSym [8] uses symbolic execution to identify timing side-channel leakage. Such techniques suffer from scalability issues on large software. In addition, CaSym and MicroWalk include only a limited scope of the micro-architecture mainly focusing on the cache. Metrics such as

TABLE I: Comparison of HASTE with state-of-the-art timing side-channel detection techniques.

	Type of Analysis	Objectives	Microarchitectural Information Incorporated
FlowTracker [6]	Static	Identify non constant time expression	None
ct-verif [5]	Static + Formal	Identify non constant time expression	None
dudect [9]	Dynamic	Identify non-constant time codes	Complete hardware microarchitecture
MicroWalk [7]	Dynamic	Identify timing difference in execution traces	Memory accesses, branch operations
CaSym [8]	Symbolic Execution	Identify cache-based side channels	Abstract cache model
HASTE	Static + Dynamic	Identify, quantify, and localize runtime timing side channel leakages	Complete hardware microarchitecture

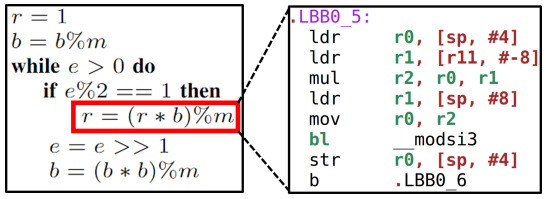

Fig. 2: The basic block in RSA modular exponentiation responsible for potential timing vulnerability.

SVF, CSV have been proposed for quantifying cache timing side channel vulnerabilities but they do not directly apply to timing vulnerabilities that arise due to secret-data-dependent control flow [10], [11].

III. The Need for Targeted Hardware-Software Analysis and Repair

The execution time of a piece of code in real hardware rarely matches hardware-agnostic estimations [12]. For example, when analyzing the RSA modular exponentiation, the basic block highlighted in Fig. 2 determines the timing difference due to the branch based on $e\%2$. If static timing analysis assumes a unit cycle model, then this is estimated as 8 cycles. For an attacker with timing *sensitivity* of 10 cycles, this system may appear secure. However, in real hardware, the multiplication (mul) may take 2.5 cycles moving the timing difference to 10.5 cycles which is within the vulnerable range for the assumed attacker. In contrast, if a static timing analysis framework assumes that all instructions takes 2.5 cycles, then the timing difference becomes 20 cycles and it may appear that the system is vulnerable to an attacker with timing *sensitivity* of 19 cycles but in reality, this is a false alarm because real hardware execution time is 10.5 cycles.

IV. HASTE Methodology

HASTE assumes that the attacker has access to the inputs of a victim program and the ability to time the overall execution (with a degree of error). A system designer can estimate an attacker's measurement error based on the level of access to the system the attacker can have. For example, remote attackers will have less timing precision compared to attackers having direct physical access. The major contribution of HASTE is a methodology for exploiting this insight into a practical detection/mitigation solution, as we describe below.

A. Hardware-Software Information Fusion

Given the software code, we first generate the control flow graph. Based on the list of assets, we perform an information flow analysis to determine the variables which are tainted by the assets. These asset-tainted variables are also treated as secrets for subsequent analysis. To precisely estimate the execution time of each basic block in the code on the specified micro-architecture, we synthesize a BOOM core that closely matches the micro-architecture. We instrument the RISC-V assembly of the code to enable cycle tracking through BOOM's hardware performance counters. This allows us to obtain the execution time (in clock cycles) for each basic block from the BOOM simulator.[1] We merge the basic block execution time estimates with the asset tainted control flow graph (previously obtained from software-level analysis), to generate a combined database for subsequent analysis.

Algorithm 1 Analyze Branch on Secret

```
1: procedure BOS(CFG, secret)
2:     Vulnerabilities =[ ]
3:     for each BB ∈ CFG do                          ▷ BB: Basic Block
4:         if branchOn(secret, BB) then
5:             PD = findPostDominatorsSet(BB, CFG)
6:             B1, B2 = Immediate branch successors
7:             Find d ∈ PD | d ∈ PostDom(B1), d ∈ PostDom(B2) and
               ∄x ∈ PD | x ∈ PostDom(B1), x ∈ PostDom(B2), d ∈
               PostDom(x), x ≠ d
8:             LP1 = LongestPath(B1, d)
9:             SP1 = ShortestPath(B1, d)
10:            LP2 = LongestPath(B2, d)
11:            SP2 = ShortestPath(B2, d)
12:            if |(LP1 − SP2)| > |(LP2 − SP1)| then
13:                BoS_Severity = |(LP1 − SP2)|
14:                Vulnerabilities.append([BoS_Severity, LP1, SP2])
15:            else
16:                BoS_Severity = |(LP2 − SP1)|
17:                Vulnerabilities.append([BoS_Severity, LP2, SP1])
18:    return Vulnerabilities
```

B. Analyzing Branch on Secret

A secret variable determining the control flow can potentially lead to timing side-channel leakage. Algo. 1 targets detection and quantification of such vulnerabilities. CFG is the asset tainted control flow graph of the given software along with the hardware-level basic block execution timing information. We first identify basic blocks which has a secret dependent branch at the end (line 4). PD is the set of post dominators of BB in CFG (line 5). $B1$ and $B2$ are the two immediate branch successors of BB (line 6). Next, we find d which is the first common post dominator of both $B1$ and $B2$ (line 7). The $BoS\_Severity$ is computed as shown in lines 13, 16. This metric is an estimate of the maximum timing difference that is possible due to a branch on secret. Hence, if an attacker can measure the execution time with a resolution less than this value then the system can be considered vulnerable.

C. Analyzing Loops on Secret

If a secret variable controls the number of times a loop runs, then an attacker can infer the variable's value by obtaining the execution time of the loop. Algo. 2 targets detection and

[1]HASTE automates the process of obtaining micro-architecture specific execution time information.

Algorithm 2 Analyze Loop on Secret

```
1: procedure LOS(CFG, secret, Sensitivity)
2:     Vulnerabilities =[ ]
3:     for each BB ∈ CFG do                          ▷ BB: Basic Block
4:         if loopHead(secret, BB) then
5:             InNode =Immediate successor of BB in Loop Body
6:             LP = LongestPath(InNode, BB)
7:             SP = ShortestPath(InNode, BB)
8:             LoS_Resilience = Sensitivity/(LP − SP)
9:             Vulnerabilities.append([LoS_Resilience, BB])
10:    return Vulnerabilities
```

Algorithm 3 Analyze Early Exit on Secret

```
1: procedure EOS(CFG, secret, Sensitivity)
2:     Vulnerabilities =[ ], L =[ ]
3:     L = findLoopsWithEarlyExitOnSecret(CFG, secret)
4:     for each (h, E) ∈ L do
5:         InNode = Immediate successor of h in Loop Body
6:         for each e ∈ E do
7:             P1 = shortestPath(InNode, e)
8:             P2 = longestPath(InNode, h)
9:             EoS_Resilience = (Sensitivity + P1)/P2
10:            Vulnerabilities.append([EoS_Resilience, h, e])
11:    return Vulnerabilities
```

quantification of these vulnerabilities. CFG is the asset tainted control flow graph of the software code with hardware-level basic block timing. First, we identify the basic blocks which are loop-heads for loops that are controlled by a secret asset (line 4). $InNode$ is the immediate successor of BB inside the specific loop body. We quantify a loop-on-secret timing vulnerability using $LoS\_Resilience$ which is computed as shown in line 8. It estimates the minimum number of iterations the loop must run before an attacker can pick up the timing discrepancy. Here $Sensitivity$ is a user-provided parameter that specifies the estimated time measurement accuracy. An attacker with physical system access may have lower $Sensitivity$ than an attacker attempting a remote analysis.

D. Analyzing Early Exit on Secret

If there is a secret variable dependent possibility for early termination of a loop then by timing the program or the loop, an attacker can potentially extract the secret. We detect and quantify such vulnerabilities using Algo. 3. L is the set of (h, E), where h is the loop head of a loop with a set of E secret-dependent early exit nodes (line 3). $InNode$ is the immediate successor of h which is also in the corresponding loop body. For each basic block (node) e in E, we compute the shortest path from $InNode$ to e (line 7) and the longest path from $InNode$ to h (line 8). To quantify an early-exit-on secret vulnerability we define $EoS\_Resilience$ as shown in line 9. Assume that in scenario-1 the loop exits after A full-iterations and in scenario-2 the loop exits after B full-iterations. If $A > B$, the extra iteration count is $A - B$. $EoS\_Resilience$ is the minimum value of $A-B$ for which an attacker (with a specific $Sensitivity$) can pick up this timing difference between the two scenarios.

V. Results & Case-Studies

To capture cycle times for different architectures, we use the configurable Berkeley Out-of-Order Machine (BOOM) [13] with a 10-stage pipeline and the Chipyard [14] framework. These two tools are used to test 5 different architectures without branch prediction as shown in Table II. The microarchitecture parameters, other than the ones that are specifically mentioned, are kept at default values. The LLVM intermediate representation (IR) of the program under test is transformed to RISC-V assembly using Clang 9.0.0. The RISC-V assembly is modified to approximately capture cycle times at the beginning and end of each basic block by wrapping the basic block with `csr mcycle` instructions. The augmented code is compiled to a RISC-V executable that is run on each architecture using the Verilator simulator.

TABLE II: BOOM Architectures used for the experiments.

Parameter	Architecture				
	Large	Medium	Small	Tiny	Mini
Fetch Width	8	4	4	4	4
Decode Width	3	2	1	1	1
Issue Width	5	4	3	3	3
ROB Entries	96	64	32	4	2
Int Register File	100	80	52	52	52
FP Register File	96	64	48	48	48
Load/Store Queue	24	16	8	8	8
D-Cache (KiB)	32	16	16	16	16
I-Cache (KiB)	32	16	16	16	16
L2 Cache (KiB)	512	512	512	512	512

TABLE III: $LoS\_Resilience$ values for the vulnerability at line 8 of the RSA modular exponentiation subroutine.

Architecture	LoS_Resilience at Different Sensitivity (S)				
	S = 5	S = 25	S = 100	S = 200	S = 300
Large	0.31	1.56	6.25	12.50	18.75
Medium	0.31	1.56	6.25	12.50	18.75
Small	0.28	1.39	5.56	11.11	16.67
Tiny	0.28	1.39	5.56	11.11	16.67
Mini	0.21	1.04	4.17	8.33	12.50

A. Analyzing RSA Modular Exponentiation

The RSA modular exponentiation subroutine shown in Algo. 4, is a classic example of vulnerable code. The direct branch on the secret keybit (e) in line 9 and the loop on e in line 8 are the two main sources of timing side-channel leakage. Using HASTE, we quantify the severity of these vulnerabilities for five different micro-architectures. In Fig. 3, we report the $BoS\_Severity$ values for the branch on secret vulnerability present at line 9. An attacker with timing sensitivity less than $BoS\_Severity$ can potentially detect which branch was taken and subsequently infer the value of e. The Large and Medium micro-architectures appear to be more resilient to this vulnerability because the execution times of basic blocks are lower leading to smaller discrepancies between LP and the SP (Algo. 1). We observe that the Small and Tiny Architecture have the same BoS Severity of 18 cycles because the additional ROB resources in Small micro-architecture were not utilized during the execution of the critical basic blocks. The Mini micro-architecture is most vulnerable due to longer basic block execution time. Based on the $BoS\_Severity$ the system designer can apply a targeted, minimal patch (path balancing) to the code to lower the metric below the attacker's $Sensitivity$. Table III reports the $LoS\_Resilience$ for the vulnerability at line 8 across different micro-architectures and $Sensitivity$ values. At $Th = 100$ cycles $LoS\_Resilience = 4.17$ (for Mini) implies that an attacker with timing measurement precision of 100 cycles can detect a timing difference arising from $ceil(4.17) = 5$ iterations of the target loop (at line 8).

Algorithm 4 RSA: Modular Exponentiation

```
1: procedure MODPOW(b, e, m)
2:     r ← Initialize
3:     if m == 0 then
4:         return 0
5:     else
6:         r = 1
7:         b = b%m
8:         while e > 0 do
9:             if e%2 == 1 then
10:                r = (r * b)%m
11:            e = e >> 1
12:            b = (b * b)%m
13:        return r
```

Algorithm 5 Binary Search

```
1: procedure BINARYSEARCH(A[], x, n)
2:     Initialize low = 0, high = n − 1
3:     while low ≤ high do
4:         mid = (low + high)/2
5:         if x == A[mid] then
6:             return mid
7:         else if x < A[mid] then
8:             high = mid − 1
9:         else
10:            low = mid + 1
11:    return −1
```

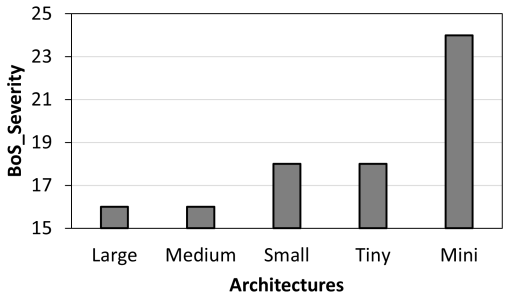

Fig. 3: $BoS\_Severity$ for the vulnerability at line 9 of RSA modular exponentiation across different micro-architectures.

B. Analyzing Binary Search

Searching algorithms are widely used in many sensitive applications such as healthcare systems and data storage systems. In a binary search (Algo. 5), the early exit at line 6 is dependent on the secret data (x) for which the search is being made. Using HASTE we report the $EoS\_Resilience$ for this vulnerability across different micro-architectures and for different $Sensitivity$ values in Table IV. With attacker measurement precision $Th = 25$, for the Large micro-architecture, we observe that $EoS\_Resilience = 2.25$. This implies that the attacker can detect a timing difference between an execution that terminates after i iterations and another execution that terminates after $i + ceil(2.25)$ iterations. Depending on how long it takes to complete the search, the attacker can potentially infer a set of candidate values for x. For this case, the system designer can minimally patch the code to reduce the timing difference until the $EoS\_Resilience$ metric is greater than the maximum number of loop iterations.

VI. Conclusion

We have presented HASTE, a joint hardware-software aware timing side-channel analysis framework. We have also formalized the detection and quantification algorithms for three different sub-classes of timing side-channel vulnerabilities. Using HASTE, we have analyzed different secure algorithms across five different micro-architectures. The ability to detect/quantify timing side-channel vulnerabilities for a specific software micro-architecture, and attacker timing measurement sensitivity combination can help system designers fix only the relevant vulnerabilities for a target hardware. This is expected to result in faster system debugging and lower performance overhead due to constant-time programming.

In future work, we will analyze systems with more microarchitectural optimizations and states, in particular cache timing side-channel attacks, and investigate automatic repair of vulnerabilities detected by HASTE.

TABLE IV: $EoS\_Resilience$ values for the vulnerability at line 6 of the binary search algorithm.

Architecture	EoS_Resilience at Different Sensitivity (S)				
	S = 5	S = 25	S = 100	S = 200	S = 300
Large	1.00	2.25	6.94	13.19	19.44
Medium	0.89	1.95	5.89	11.16	16.42
Small	0.75	1.38	3.72	6.84	9.97
Tiny	0.58	0.95	2.31	4.13	5.95
Mini	0.67	0.90	1.80	2.99	4.18

VII. Acknowledgement

This work was funded in part by Semiconductor Research Corporation (SRC), task 2860.001.

References

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.
[2] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.
[3] D. J. Bernstein, "Cache-timing attacks on aes," 2005.
[4] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 431–446.
[5] 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.
[6] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 110–120.
[7] 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*, 2018, pp. 161–173.
[8] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 505–521.
[9] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1697–1702.
[10] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 106–117.
[11] T. Zhang, F. Liu, S. Chen, and R. B. Lee, "Side channel vulnerability metrics: the promise and the pitfalls," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013, pp. 1–8.
[12] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.
[13] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," May 2020.
[14] A. Amid *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.