Simulating Our Way to Safer Software: A Tale of Integrating Microarchitecture Simulation and Leakage Estimation Modeling

Justin Feng, *Graduate Student Member, IEEE*, Fatemeh Arkannezhad, *Graduate Student Member, IEEE*, Christopher Ryu, *Student Member, IEEE*, Enoch Huang, *Student Member, IEEE*, Siddhant Gupta, *Student Member, IEEE*, and Nader Sehatbakhsh, *Member, IEEE*

***Abstract*—An important step to protect software against side-channel vulnerability is to rigorously evaluate it on the target hardware using standard leakage tests. Recently, leakage estimation tools have received a lot of attention to improve this time-consuming process. Despite their advancements, existing tools often neglect the impact of microarchitecture and its underlying events in their leakage model which leads to inaccuracies. This paper takes the first step in addressing these issues by integrating a physical side-channel leakage estimation tool into a microarchitectural simulator. To achieve this, we first systematically explore the impact of various architecture and microarchitecture activities and their underlying interactions on the produced physical side-channel signals and integrate that into the microarchitecture model. Second, to create a comprehensive leakage estimation report, we leverage taint tracking and symbolic execution to accurately analyze different paths and inputs. The final outcome of this work is a tool that takes a binary and generates a leakage report that covers architecture and microarchitecture-related leakages for both data-dependent and path-dependent information leakage scenarios.**

***Index Terms*—Leakage modeling, side-channels.**

I. Introduction

In developing reliable and trustworthy software and systems, physical side-channel signals (e.g., electromagnetic and power) pose an important security concern [1], [2]. Thus, side-channel attacks can be prevented by estimating the potential leakage and patching the software properly during its design.

A popular approach for side-channel leakage estimation relies on automated modeling tools [3], [4], [5], [6], [7], [8]. These tools *emulate* the underlying system and hardware and accurately model the interaction between the hardware and software. For a given application, such models provide a software tool for estimating the physical side-channel leakage.

This paper improves the state-of-the-art by proposing a new comprehensive modeling tool called Gloria. The ***key contribution*** is that our model is able to automatically identify leakage scenarios for three different layers: architecture, microarchitecture, and events – the third one has been absent in all existing tools, as well as modeling both data-dependent and path-dependent leakage scenarios.

Manuscript received 4 July 2023; accepted 1 August 2023. Date of publication 10 August 2023; date of current version 31 August 2023. This work was supported in part by NSF under Grants CNS-2211301 and CNS-2312089. *(Corresponding author: Nader Sehatbakhsh.)*

The authors are with the School of Engineering, Department of Electrical and Computer Engineering, University of California, Los Angeles, CA 90095 USA (e-mail: jfeng10@ucla.edu; fatemeharkan@g.ucla.edu; cryu17@g.ucla.edu; eihuang@g.ucla.edu; siddhantgupta@g.ucla.edu; nsehat@ucla.edu).

Digital Object Identifier 10.1109/LCA.2023.3303913

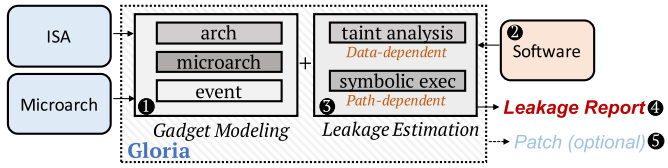

Fig. 1. The steps for designing our leakage estimation tool, Gloria. Our tool first builds a model from all possible instructions (RISC-V) and system events. Using this model, it extends a microarchitecture simulation tool to enable leakage estimation. We then develop new features for the tool to find data and path-dependent leakages and automatically generate a comprehensive leakage report. Optionally, it can patch the software based on the report.

To achieve this, our method is designed in two main steps as outlined in Fig. 1. The fundamental idea is to integrate a leakage estimation model into a microarchitecture simulation tool. This is implemented by first, systematically searching different instruction sequences to identify *"leakage gadgets."* Briefly, we define a leakage gadget as any sequence of instructions where changing the input would create a statistically significant signal using a distance metric (e.g., Hamming distance). Compared to prior work, however, this search includes architecture/microarchitecture (i.e., sequence of instructions and their underlying interactions) AND microarchitecture events (e.g., cache misses, branch mispredictions). The latter is achieved since a detailed microarchitecture simulator is embedded in the tool.

The second step (2 and 3 in Fig. 1) is to design an analysis tool that takes the list of leakage gadgets and a binary, searches through the binary, and lists all detected leakage sequences. The tool then generates a report that could be used for modifying/repairing the software to eliminate these gadgets and ultimately removing the side-channel leakage scenarios. Gloria reports the leakage in two categories: *data-dependent* leakage and *path/control-flow-dependent* leakage. To ensure high code coverage and accurate data-flow tracking, Gloria employs symbolic execution [9] and dynamic taint tracking [10].[1] The final outcome is a report for data and path-dependent leakages (4 in Fig. 1).

To measure the signals for each given instruction sequence in step 1, Gloria utilizes *real hardware*. Specifically, we implement an open-source in-order RISC-V processor and measure the side-channel (power and/or EM). The microarchitecture is based on *marss-riscv* simulation tool [11]. Further, we evaluate

[1]The analysis could be further strengthened by employing fuzzing which is left for future work.

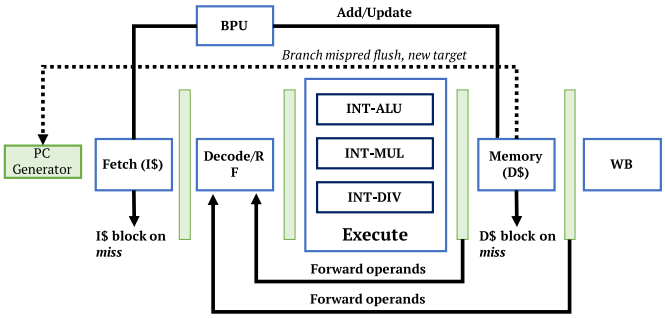

Fig. 2. The microarchitecture used in our setup for measuring the real side-channel signals based on *marss-riscv* [11]. The processor has five stages of (in-order) pipeline and supports RISC-V 32-bit ISA (RV32GC).

our model using a standard set of benchmarks. Specifically, we use Mibench suite [12] which is a standard and popular set of tools for embedded systems applications.

In short, the main contributions of this paper are as follows:

- The design and implementation of a new comprehensive leakage estimation mechanism that models the system at three levels including the system event level.
- The development of an automated tool to find all sequences of instructions that could create side-channel leakage for a given program.
- A proof-of-concept implementation of the tool using open-source simulation tools and standard benchmarks.

II. Design

Overview: The key to our design is creating a fully customizable yet realistic measurement setup. To achieve this, we rely upon developing a RISC-V-based processor. We implement our design on an FPGA to collect real signal traces using an EM probe placed close to the FPGA board.

The microarchitecture of our design is shown in Fig. 2. Our design is *heavily based on the microarchitecture used in the MARSS-RISCV simulation tool* [11], as GLORIA is built on top of this simulator. With this method, we can measure both physical signals from real hardware (using our FPGA implementation) and events using the MARSS simulator. Note that to build our leakage estimation tool, we assume that the designer has full knowledge of the microarchiteture.

Leakage Modeling: Our framework, GLORIA, uses a multi-step algorithm to estimate leakage. The ultimate goal of the tool is to take a binary as input and outputs a report that describes various leakage scenarios. The details of our algorithm are shown in Algorithm 1. Briefly, the algorithm first estimates the leakage for each instruction. Then, it computes the pairwise information leakage (i.e., the difference between a pair of instructions) using a metric. GLORIA then extends this model further to consider system events. This is achieved by repeating the pairwise experiments but this time by intentionally creating a system event (e.g., a cache miss) during that measurement. Combining these together, GLORIA is then able to take an assembly program as input and parse through the program, and estimate the leakage for each instruction and for the overall program. This will be reported for both data-dependent leakage (i.e., same instruction, different data) and path-dependent leakage (i.e., different instructions).

Algorithm 1: Calculating Architecture, Microarch, and Event-Level Leakages for All Groups of Instructions.

1 $ISA\_set = \{NOP, ALU, LD, Shift, Mul, Div, Br, ST\}$
  $Reg\_set = \{[x0, x1, x2], [x15, x16, x17]\}$
2 $Post\_set = Pre\_set = \{NOP, NOP, NOP, NOP, NOP\}$
3 *Initialize HD, HW, DHW, PHW, EHW*
4 **for** $i = 0$ *to* 7 **do**
5 $Set = \{Pre\_set, ISA\_set[i], Post\_set\}$
6 **for** $j = 0$ *to* 1 **do**
7 $Prog =$ *instGen(Set, Reg_set[j])*
8 $HD[i][j] =$ *sigPower(exec(Prog))*
9 `// Data-dependent leakage:`
10 **for** $i = 1$ *to* 7 **do**
11 $DHW[i] = |HD[i][0] - HD[i][1]|$
  `// Path-dependent leakage:`
12 **for** $i = 1$ *to* 7 **do**
13 **for** $j = 1$ *to* 7 **do**
14 **for** $k = 0$ *to* 1 **do**
15 **for** $t = 0$ *to* 1 **do**
16 $cur = |HD[i][k] - HD[j][t]|$
17 **if** $cur > PHW[i, j]$ **then**
18 $PHW[i, j] = cur$
19 $Event\_set = \{Mem, Pred, Trap, Haz, Funct, Forw\}$
  `// Potential list of events.`
20 **for** $i = 0$ *to* 5 **do**
21 $Prog =$ *instGenE(Event_set[i])*
22 $HDE[i] =$ *sigPower(exec(Prog))*
23 **for** $j = 1$ *to* 7 **do**
24 $EHW[i][j] = |HDE[i] - HD[i][j]|$
  `// Vectors will be padded with zeros if not the same size.`

Step 0: Initialization. The design of GLORIA starts with a pre-processing phase. During this phase, we first decide what group of instructions and events need to be considered. Further, we need to consider what types of data-dependent/register-level activities should be modeled.

To achieve this, we first focus on the instruction modeling problem. One approach to consider is to model *all* instructions (and their grouping) for the given ISA. While this would be quite accurate, it comes with a large computational complexity. For RV32GC, there are more than 60 unique instructions, therefore, for a five-stage pipeline design, the total number of possible groupings would be close to one trillion unique combinations. Clearly, this makes the measurement cumbersome, thus some clustering is needed. Similar to prior work [5], [7], we observe that individual instructions can be formed into *supergroups* where, intuitively, instructions in each group excite similar parts of the hardware and hence have very similar side-channel signals. As can be seen in line 1 of Algorithm 1, we observed

that there are eight different distinct groups to consider in our design.

Step 1: Baseline Leakage Computation. Once the pre-processing phase is completed, we will start computing the leakage for each supergroup by creating an instruction sequence that consists of the target instruction padded (before and after) with NOP (no operation) instructions. This will ensure that the interferences caused by other units are minimal.

The created sequence is then passed to *instGen* function, which takes the sequence and *register set* (i.e., "high" or "low"), and creates an instruction sequence, *Prog*, that can be executed on the processor. "high" is for generating lots of bit flips while "low" is the opposite. Note that we also considered instructions with register-immediate as operands and observed that they behave quite similarly to register-register operations with the same activity (e.g., ADD vs. ADDI).

The generated program is executed on the real hardware and the corresponding signal is collected and stored. The Hamming distance (HD) for each signal when compared to a signal generated by an ALL NOP sequence (baseline) is calculated for all supergroups (for both "high" and "low").

Step 2: Data and Path-Dependent Leakage Estimation. Once the leakages over the baseline were computed, we extends the method to compute what we call the *pair-wise leakages*. We use this insight that information can be leaked through side-channels by two scenarios: ***data-dependent*** and ***path-dependent leakage***. In both cases, an adversary observes/collects multiple runs and by analyzing the differences, infers some secret.

In the *path-dependent* scenario, information is leaked because the program follows different paths. Since each path executes a unique sequence of instructions, the generated side-channel signal may exhibit variations. An adversary gathers multiple signals and notices their dissimilarity at one or more locations, suggesting non-similarity in the paths taken for each occurrence. If execution is dependent on a secret value, the detection of these differences will reveal the secret value(s).

In the *data-dependent* scenario, the code follows the same path but with different input values (i.e., different operands, register and/or immediate, for each instruction). The adversary collects multiple runs and analyzes them. If any difference can be observed, the adversary can find the secret input(s).

To compute the leakage in each scenario, GLORIA follows lines 9-18 in Algorithm 1. Specifically, it computes two weights, path and data Hamming weight, *PHW* and *DHW*. The data-dependent leakage (*DHW*) is computed by comparing the low-high runs for each given supergroup. The path-dependent leakage is a pair-wise metric (i.e., an 8×8 matrix). Since each pair has four possible values (i.e., all possible pairings of low-high), we keep the *maximum* (see lines 17-18 in Algorithm 1).

Step 3: Event Leakage Estimation. The final and critical part of GLORIA is adding the estimation for system events. Recall that our goal is to model various events. Line 19 in Algorithm 1 defines this list. We consider *six* different events. It is important to mention that we only selected events that could be handled in our implementation (see Fig. 2). Additional events (e.g., page fault, system call) could be modeled by adding additional support to our design. We believe the current list is realistic although not comprehensive.

To estimate leakage, each event is passed to a function named *instGenE*. This function is responsible to create a sequence (aka program) that can generate the desired event (e.g., a cache miss). Note that this can be achieved by controlling the hardware implementation (e.g., we can control when and how a particular event such as cache miss can happen by adding additional control signals). We carefully change the hardware when needed. It is important to note that the changes were minimal to avoid changing the baseline signals.

Similar to step 2, the program is then executed and the signals are collected. The event-related leakages are calculated and stored in a matrix (6 events and 7 supergroups) using the same distance metric explained before.

System Integration: To analyze a program with the leakage model, we extend the baseline microarchitecture simulator with the leakage estimation using the method explained above. We now turn our attention to systematically analyzing a binary to extract all leakage scenarios.

GLORIA leverage binary analysis tools, specifically *angr* [9], to extract two critical pieces of information. First, we extract a list of all basic blocks for a given program. Second, a set of inputs for each unique path that has been covered by *angr*'s symbolic execution engine. GLORIA takes these numbers and starts its leakage analysis as described below:

1) Path-Dependent Leakage Estimation: GLORIA takes two basic blocks (i and j) as input and computes the leakage using the path and event matrices (see lines 18 and 24 in Algorithm 1). We call this $BBS[i][j]$ (basic block leakage score). Since the goal is to estimate the leakage for different paths, we don't need to compute BBS for all possible basic block pairs. Instead, only pairs that are connected by a parent basic block need to be evaluated. BBS is calculated by parsing through instructions for each basic block and computing the pairwise difference between the two instructions (path, PHW, and/or event Hamming weight, EHW). For basic blocks with different sizes, we pad the shorter ones with NOPs.

2) Data-Dependent Leakage Estimation: For a given trace (sequence of executed instruction), GLORIA also computes an overall leakage which we call PS (program leakage score). PS is essentially the summation of all executed basic blocks for a given input. For calculating this score, however, we only consider data and event Hamming weights as we are only concerned about data-dependent leakages. PS provides an estimation of how vulnerable the program is against changes in input (even when the path is identical).

Furthermore, GLORIA can support selective data-dependent leakage estimation via *taint analysis*. More specifically, instead of computing the overall leakage for all variables, GLORIA can track user-defined variables (i.e., sensitive values/inputs), and compute the potential leakage *only* for instructions that use this sensitive value and/or its dependencies (e.g., variables that are dependent on the sensitive value). This will allow a more realistic leakage estimation as only the leakage relevant to sensitive values is reported.

III. EVALUATIONS

We collected signals from real hardware using the microarchitecture design shown in Fig. 2. The hardware was implemented in Verilog HDL on an Intel Cyclone V FPGA. Intel Quartus was used for synthesis. The FPGA board we used was a Terassic DE0 board that includes an FPGA board and several other modules.

We collected the electromagnetic (EM) side-channel signals from the board using an EM probe (Langer EMV H-Field Probe).

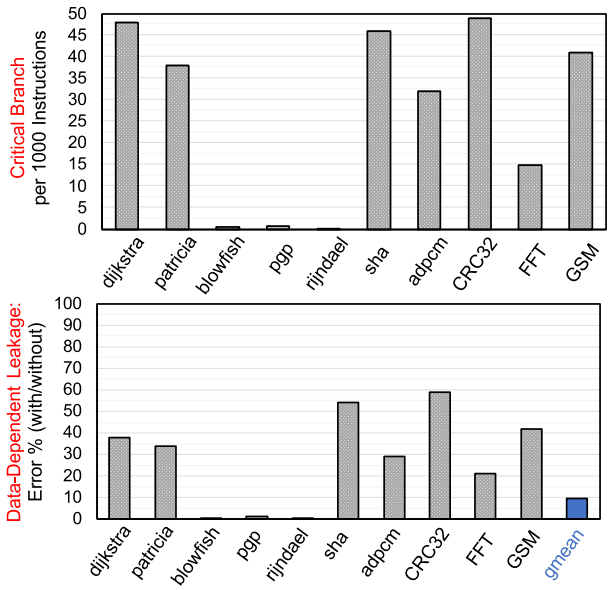

Fig. 3. Path and data-dependent leakages for Mibench benchmark. Without properly modeling the events, there will be more than 10% modeling error.

The probe was placed right on top of the FPGA (not touching) in the center of the board. We used a Keysight Oscilloscope (DSOX6002) to record the signals. The recorded signals are downsampled to have four samples/data points per cycle. We ran each recording 100 times and aligned and averaged the recordings. For aligning, we used an external trigger for the Oscilloscope. The trigger was connected to one of the FPGA pins. We modified our processor's code to generate the trigger at the beginning of each measurement.

Once the raw signals are collected for all pairings including architecture, microarchitecture, and events, our code, implemented in Python, followed the steps in Algrotihm 1 to generate DHW, PHW, and EHW. This measurement campaign requires about (order of) 100 thousand measurements which took place in a span of a month.

For computing the data-dependent and path-dependent leakages using the matrices calculated above we used Mibench standard benchmark suite [12], a popular benchmark for embedded and IoT applications.

We used MARSS-RISCV cycle-accurate simulation tool to run our applications [11]. MARSS leverages an emulator, TinyEMU, to emulate the system events and activities.

Using GLORIA, we report two results. First, in Fig. 3 (top) we report the number of critical branches (branches whose BBS score is high) that our tool has found in each application. The results are reported as the number of critical branches per 1 k executed instructions.

As can be seen in the figure, most applications have quite a lot of critical branches. The statistics are significantly better for security applications (e.g., 'blowfish'). The main reason is that these applications are inherently leveraging less number of branches with critical values.

The effect of modeling the system events is shown in Fig. 3 (bottom). Here we report the error when events are not modeled. Error is defined as the ratio of missed critical branches over all detected branches. The results are shown in percentages. As can be seen in the figure, without modeling the events, GLORIA has more than 10% modeling error on average (i.e., missing at least 1 out of 10 critical branches).

The important *takeaway* from this experiment was to emphasize the importance of leakage modeling at the early stages of software design. Proper design choices such as avoiding critical branches (e.g., 'blowfish') could significantly improve leakage estimation and allow better, low-overhead defense solutions. Such early analysis is only possible if an accurate and capable yet user-friendly tool such as GLORIA exists.

IV. CONCLUSION

A new comprehensive software leakage estimation tool was presented in this paper. The key idea was to take a systematic approach by integrating a new leakage model into an existing microarchitecture simulator.

Overall, GLORIA improves the state-of-the-art in modeling accuracy by adding the capability for modeling events. Further, GLORIA allows for detailed analysis such as detecting critical branches and/or selective secret leakage analysis.

REFERENCES

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems," in *Proc. Annu. Int. Cryptology Conf.*, 1996, pp. 104–113.
[2] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2003, pp. 13–28.
[3] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "ROSITA: Towards automatic elimination of power-analysis leakage in ciphers," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–17.
[4] M. A. Shelton et al., "Rosita: Automatic higher-order leakage elimination from cryptographic code," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2021, pp. 685–699.
[5] D. McCann, E. Oswald, and C. Whitnall, "Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages," in *Proc. USENIX Conf. Secur. Symp.*, 2017, pp. 199–216.
[6] A. de Grandmaison, K. Heydemann, and Q. L. Meunier, "ARMISTICE: Microarchitectural leakage modeling for masked software formal verification," *IEEE Trans. Comput.-Aided Design Integrated Circuits Syst.*, vol. 41, no. 11, pp. 3733–3744, Nov. 2022.
[7] N. Sehatbakhsh, B. B. Yilmaz, A. Zajic, and M. Prvulovic, "EMSim: A microarchitecture-level simulation tool for modeling electromagnetic side-channel signals," in *Proc. IEEE Int. Symp. High Perfor. Comput. Architecture*, 2020, pp. 71–85.
[8] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem, "COCO: Co-design and co-verification of masked software implementations on CPUs," in *Proc. USENIX Conf. Secur. Symp.*, 2021, pp. 1469–1468.
[9] Y. Shoshitaishvili et al., "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 138–157.
[10] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 482–493, 2007.
[11] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A Full System Simulator for Multicore x86 CPUs," in *Proc. 48th Des. Automat. Conf.*, 2011, pp. 1050–1055.
[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE 4th Annu. Int. Workshop Workload Characterization*, 2001, pp. 3–14.