FIFA: Exploring a Focally Induced Fault Attack Strategy in Near-Threshold Computing

Prabal Basu, Chidhambaranathan Rajamanikkam, Aatreyi Bal, Pramesh Pandey, Trevor Carter, Koushik Chakraborty, and Sanghamitra Roy

***Abstract*—In this letter, we explore the emerging security threats of near-threshold computing (NTC). Researchers have shown that the delay sensitivity of a circuit to supply voltage variation tremendously increases, as the circuit's operating conditions shift from traditional super-threshold values to NTC values. As a result, NTC systems become extremely vulnerable to timing fault attacks, jeopardizing trustworthy computing. Inspired by the operation of a polymorphic virus, we propose a novel threat model for NTC, referred to as a focally induced fault attack (FIFA). FIFA employs a machine learning framework to ascertain the circuit vulnerabilities and generates targeted software modules to cause a breach of end-user privacy. Our experimental results, obtained from a rigorous machine learning approach, indicate the efficacy of FIFA, in a low-power mobile platform.**

***Index Terms*—Computer security, fault diagnosis, system-on-chip.**

I. Introduction

COMPUTING is on an upward trend of integrating with humans and our environment through an interconnected network of wearables, digital dust, smartphones, and desktops. With this upsurge, we observe two fundamental trends in low power computing that are poised to reshape the trustworthy properties of future hardware platforms. First, there is a dramatic rise of low power systems and applications [e.g., Internet of Things (IoT)] that are systematically intertwined into our mere existence. Many IoT systems, implantable medical sensors, for example, have stringent constraints on power consumption, while demanding ever growing computing power and functional diversity. Second, hardware researchers are actively engaged in exploring a new generation of device and circuit technologies that can offer massive improvements in the energy efficiency of computation. In this ever-growing demand for energy efficiency, near-threshold computing (NTC) has emerged as one of the promising directions [1]. While many recent works have explored the challenges and opportunities of reliable computing at near-threshold [2], the *security vulnerabilities* spawning from such low-power computing frameworks have received only a marginal attention.

This work was supported by the National Science Foundation under Grant CAREER-1253024, Grant CCF-1318826, Grant CNS-1421022, and Grant CNS-1421068.

The authors are with USU BRIDGE LAB, Electrical and Computer Engineering Department, Utah State University, Logan, UT 84322 USA (e-mail: prabalb@aggiemail.usu.edu; chidham@aggiemail.usu.edu; aatreyi.bal@aggiemail.usu.edu; pandey.pramesh1@aggiemail.usu.edu; trevor.carter@aggiemail.usu.edu; koushik.chakraborty@usu.edu; sanghamitra.roy@usu.edu).

NTC circuits feature a tremendous delay sensitivity to supply voltage variation [3], enabling ample opportunities for timing fault attacks [4]. Recently, Bal *et al.* [5] have shown that a small number of process variation (PV) affected logic-gates at NTC, can alter the critical paths, sensitized during a program execution, which was unprecedented in the realm of traditional super threshold computing. Moreover, the PV-induced delay variability is substantially different in various NTC chips of the same design [3]. Considering these trends, we envision a threat model, referred to as a focally induced fault attack (FIFA), that exploits the circuit-level variability of the NTC systems, to inflict security threats at the application-level. FIFA employs a machine learning framework to design a malicious software module. The malicious module, analogous to a polymorphic virus, learns about the circuit-level vulnerabilities of individual victims—NTC processors—and subsequently uses the knowledge to launch targeted fault attacks. Specifically, we make the following key contributions in this letter.

1) We propose a novel threat model—FIFA—that exploits the emerging security vulnerabilities of the NTC circuits to inflict potent timing fault attacks in low-power mobile platforms (Section II).
2) We develop a machine learning framework, called *SmartLearn*, to identify the circuit-layer vulnerabilities and generate software modules to cause a breach in security (Section III).
3) Using the *SmartLearn* framework, we ascertain the vulnerable instructions for various NTC chips of the same design (Section V).

II. Proposed Threat Model: FIFA

Fig. 1 illustrates our threat model, FIFA, in the realm of a low-power mobile platform. FIFA is inspired by a fundamental principle of a viral attack in a biological system, where a virus is able to mutate itself during its transmission from one human host to another, thereby vastly improving its efficacy in different hosts with diverse immunities. Fig. 1 shows the boundary of trust in FIFA. We consider that a rogue software developer will stealthily insert a malware in an application software. This malware learns the specific vulnerabilities of its victim processor, and subsequently morphs itself to launch potent fault attacks.

A. Phases of the FIFA

We discuss three operational phases of FIFA in order to establish its practicality.

1) Malware Insertion: The malware can be inserted in the application software in two different ways. First, a few rogue application developers can compromise the rigor of

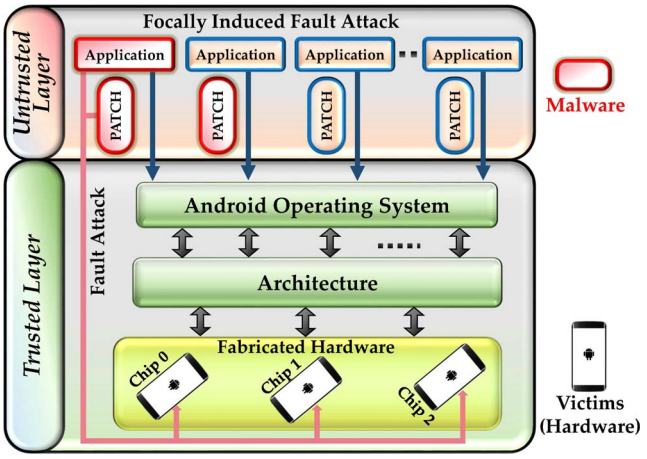

Fig. 1. Proposed FIFA threat model showing the trusted and untrusted components in a low power mobile platform.

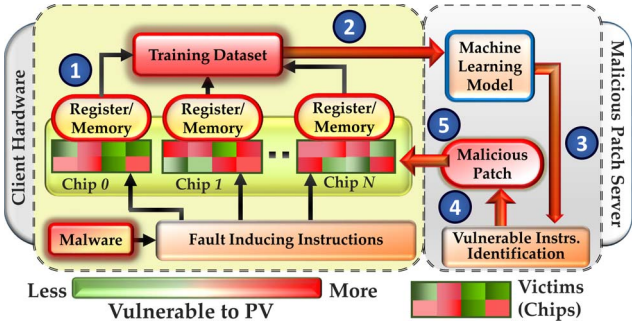

Fig. 2. Various operational stages of an FIFA using SmartLearn. Five distinct phases of FIFA are marked as: (1) vulnerability diagnosis at the chip level; (2) vulnerability data transfer to remote server; (3) processing diagnostic data using SmartLearn; (4) customized malicious patch creation; and (5) patch delivery to its target chip.

the software testing process, and introduce a malware in the initial release of the software. The malware, with a low footprint, can efficiently disguise in the statistics collection module of the software that collects user-approved logs for offline analysis [6]. Second, the initial software release can be malware-free, but subsequently, it may be tampered through a malicious patch [7]. The modified application can then launch an FIFA, compromising the integrity of the mobile platform. Many users, nowadays, download and install updates from untrusted repositories, which intensifies the scope of a malware insertion through unreliable, malicious patches [8].

2) Activation: An existing bug in the legacy code of the application software can trigger the malware by creating a rare event [9]. We ensure a covert FIFA, by using a time-based trigger, similar to the cron job in a Unix-based operating system. Moreover, the diverse PV signatures of the NTC chips and the corresponding delay characteristics prevent an early detection of the malware.

3) Operation: The malware adopts a two-pronged strategy: 1) diagnose and learn the vulnerabilities of the victim processors and 2) launch targeted FIFAs exploiting the vulnerabilities.

1) *Learning Vulnerabilities:* Different chips of the same design can have different delay profiles due to PV, signifying diverse vulnerabilities. The malware comprises a low-overhead diagnostic module that learns the vulnerable instructions of various victim processors by inducing timing faults in the victims from the software layer. The malware uses a software-implemented fault injection technique to induce timing faults (details in Section III-C). The malware then offloads the vulnerable instruction data, consolidated from many victim processors, to a remote server. Considering multiple remote servers prolongs the eradication of the attack, thereby fortifying our threat model.
2) *Launching FIFAs:* At the remote server, intelligent machine learning algorithms are used to identify a small, yet potent, set of vulnerable instructions, that can harm a large majority of the victim processors. The malware then creates a malicious patch that launches targeted FIFAs, when installed during a software update process in the victim mobile phone processors.

III. SmartLearn: Cross-Site Learning Framework

Fig. 2 illustrates different components of our proposed learning framework—*SmartLearn*. *SmartLearn* aims to automatically learn the vulnerabilities of various chips of a design, to facilitate an FIFA. Using a cross-site collaborative framework, *SmartLearn* exchanges information between the malicious application server, and the victimized client hardware. We outline the design challenges of *SmartLearn* (Section III-A), and discuss the learning algorithm (Section III-B).

A. Challenges of SmartLearn

Creating a rogue learning infrastructure faces a couple of unique design challenges, outlined next.

1) Colossal Search Space: An attacker needs to retrieve the vulnerabilities of millions of chips of a design, in order to create a malicious patch, equally detrimental for a majority of the chips. As the vulnerabilities vary across the chips, designing a potent customized attack for each chip would extend the search space to mammoth proportions. Hence, it is imperative to optimize the search space for a few consolidated patches, that would be maximally damaging for the largest possible share of the chips.

2) Vulnerability Diagnosis: An effective FIFA design involves an accurate assessment of the hardware-layer vulnerability through the software-layer. It is extremely difficult for an application software, to precisely inject a timing fault in the underlying circuit, crucial for vulnerability detection. Moreover, ensuring a stealthy operation requires the malicious software to maintain a low activity footprint.

B. Design of SmartLearn

To overcome the design challenges, *SmartLearn* employs a holistic *Machine Learning* approach, comprising four stages: 1) software-induced fault generation; 2) training data accumulation; 3) machine learning model development; and 4) patch formation. Stages 1)–3) correspond to steps 1–3 in Fig. 2, respectively. Stage 4) corresponds to steps 4 and 5, in Fig. 2. Each of these stages is described next.

C. Software-Induced Fault Generation

We assume that the vulnerable instructions of a (post-silicon) validated chip cannot deterministically cause timing violations. This is a reasonable assumption as timing violations are highly contingent on complex sequencing of opcodes

and operands, and determining such sequences is nontrivial [10]. Hence, the malware needs to create an environment, that induces timing violations during an instruction execution. In this pursuit, we instrument the malware with two distinct categories of instructions, described next.

1) Fault Inducing Instructions: Bertran *et al.* [11] have shown that consecutive executions of a sequence of high power consuming instructions and a sequence of low power consuming instructions induce a voltage droop in the power delivery network of the core. Collectively, these two sequences of instructions are called a *stressmark*. The induced voltage droop can increase the instruction delays, potentially causing a timing fault within a pipeline stage [12]. Due to an aggravated delay sensitivity to supply voltage variation at NTC, the execution of these fault inducing instruction sequences makes the next few *benign* instructions in the pipeline vulnerable to timing errors. We consider multiple distinct droop levels to enforce a timing error for a given instruction (Section III-D). Different levels of voltage droop can be precisely generated at the software level using well-known stressmark generation methodologies [13], [14]. As the stressmark parameters are ascertained at the remote server, the associated overheads do not impact the stealth of the malware.

2) Vulnerable Instructions: These instructions contain a subset of all the opcodes supported by the ISA of the victim processor. For a given instruction, we examine the impact of the operand length on the occurrence of timing errors. To estimate the vulnerability, we pair each opcode with operands of varied lengths, and observe the extent of violations for such pairs. We assume an instruction to be vulnerable, only if it latches an incorrect data in the register file, after the execution of the fault inducing instruction sequence. Identifying the vulnerable instructions is done in multiple short intervals to maintain the stealth of the malware.

D. Training Data Accumulation

In the *SmartLearn* framework, the vulnerable instructions, along with a few metadata comprise the features of our training data set. A few features that we study, are: opcodes, number of operands, input operand widths, multiple droop levels for fault injection, and the number of bit flips in the pertinent register file entry. The malware reads the register values after the execution of the intended instruction, and compares it against the correct stored values, to infer a timing error. The violated instructions and the aforementioned metadata are stored in the application's memory space, in the form of a hash-table. Considering 34 opcodes of the ARM ISA, with each instruction being 32 bits long, and 8-bit entries to hold the number of bit-flips per instruction for three droop levels, the size of the hash-table is approximately 250 bytes.

E. Machine Learning Model Development

A machine learning model is developed at the server, to analyze the diagnostic information on vulnerable instructions, collected from millions of chips of a given model, and to create a few code patches that offer potent attacks to a majority of these chips. In this letter, we compare the performances of multiple machine learning algorithms, viz., support vector machine, XGBoost, naive Bayes classifier, and random forest, on our training data set. The details on features selection, model accuracies and determination of the vulnerable instructions are discussed in Section V.

F. Patch Formation

Upon building the model, we evaluate its accuracy using tenfold cross validation. Subsequently, we rank the opcodes based on their inferred vulnerabilities and determine what operands are suitable to create the most vulnerable instructions (Section V-B). These vulnerable instructions are then included in the malicious patch. The patch, when installed during a software update process, launches FIFAs in the client hardware.

Please note that generating a distinct patch for every chip is not realistic in our threat model. To ensure a covert operation, the malware collects and offloads only a minimum diagnostic information for each victim, to the remote server (Section III-D). Deploying customized patches to the appropriate clients requires the signatures of the clients. Transmitting such signatures would incur additional overhead, thwarting the stealth of the FIFA. Hence, we resort to generating a single consolidated patch, detrimental for a large share of the victim chips.

IV. Methodology

We customize the PTM 14-nm technology model cards for HSPICE, to study the comparative delay distributions for the basic gates at the NTC region [15]. To model within-die process-variation, we use the VARIUS-NTV setup [3]. We synthesize the RTL of the execute pipe-stage of a 32-bit FabScalar generated core [16], with the 15-nm Nangate library [17], using the Synopsys Design Compiler. The operating voltage and frequency at NTC are assumed to be 0.45 V and 250 MHz, respectively.

We use our in-house statistical timing analysis (STA) tool to study the timing errors of all the sensitized paths of the execute stage, for seven different arithmetic and logic operations [5], [18]. The operands are chosen to cover a typical range seen during the executions of real applications. The STA tool uses a library of delay files for the basic gates, at different operating conditions. The library comprises both PV-affected and PV-free delay information for all the gates. The inputs to the STA tool are the synthesized netlist of a circuit and the input vectors corresponding to the netlist. The input vectors can be generated from an architectural simulator.

V. Experimental Results

In this section, we present the vulnerable instructions, along with their relative vulnerabilities, obtained from our proposed machine learning framework—*SmartLearn*.

A. Feature Selection

Determining the set of most vulnerable instructions can be categorized as a classification problem. For each training instance, we calculate a weighted-average of the *number of bit-flips*, induced by various droop levels during an instruction execution. We assign a higher weight to the low-droop induced bit-flips, to signify a more vulnerable opcode. If the weighted-average is more than a preset threshold value, we infer the instruction to be vulnerable. We use the *feature_importances_* method of the random forest classifier (in scikit-learn [19]) to abandon *the number of operands* as a feature due to a very poor importance. Finally, we use the *opcode type*, *first operand width* and *second operand width*, as the features for training. The training set comprises 100 000 instances.

TABLE I
AVERAGE ACCURACIES FROM TENFOLD CROSS VALIDATION

Classifier	Average Accuracy
Random Forest	85.5%
Support Vector Machine	85.4%
Naive Bayes	85.1%
XGBoost	85.5%

B. Ranking Vulnerable Instructions

Table I reports the average accuracies of various classifiers from tenfold cross validation, using the scikit-learn Python package [19]. The average accuracies for other *folds* (e.g., 5–9) vary within 1% compared to those reported in Table I. Subsequently, we rank the opcodes based on their inferred vulnerabilities, using [20, eq. (3)]. We consider seven opcodes in our experiments. In the decreasing order of vulnerability, the rank of the seven opcodes are: AND, MULT, LOAD, ADD, SUB, XOR, and OR. We have also found that a larger operand has a greater contribution to the vulnerability of an instruction. So, we conclude that the top ranked opcodes, combined with large operands, are favorable to launch FIFAs in a large share of chips of the same design.

VI. RELATED WORK

A. Fault Attacks

Fault Attacks leak secure information from a circuit by modifying the circuit's operating environment in order to change its expected behavior. Fault attacks can be categorized into safe error, algorithm modification, and differential fault analysis (DFA). Safe error attacks distinguish between normal and abnormal behaviors of the chip to retrieve sensitive information [21]. Balasch *et al.* [22] demonstrated algorithm modification, by replacing the instructions executed by a microcontroller. On the other hand, DFA extracts the cryptographic keys by comparing the original and faulty ciphertexts [23]. In this letter, we exploit the aggravated timing fault vulnerability of the NTC circuits to propose a novel threat model, called FIFA.

B. Side Channel Attacks

Side channel attacks (SCA) harness the information leakage from the physical implementation of a hardware. Among the recent works, Liu *et al.* [24] have presented the efficacy of SCA in last-level caches. Our threat model, aimed at NTC circuits, may not be identified from side-channel analysis due to its stealth.

VII. CONCLUSION

In this letter, we propose FIFA—a novel threat model for NTC. FIFA employs a machine learning framework, called *SmartLearn*, to identify the vulnerabilities of many NTC circuits of a given design. By learning the vulnerabilities, FIFA creates software patches to inflict potent timing fault attacks in the victimized chips.

ACKNOWLEDGMENT

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. N. Mudge, "Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits," *Proc. IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.

[2] X. Zhang *et al.*, "Characterizing and evaluating voltage noise in multi-core near-threshold processors," in *Proc. ISLPED*, Beijing, China, 2013, pp. 82–87.

[3] U. R. Karpuzcu, K. B. Kolluru, N. S. Kim, and J. Torrellas, "VARIUS-NTV: A microarchitectural model to capture the increased sensitivity of manycores to process variations at near-threshold voltages," in *Proc. DSN*, Boston, MA, USA, 2012, pp. 1–11.

[4] B. Yuce, N. F. Ghalaty, and P. Schaumont, "TVVF: Estimating the vulnerability of hardware cryptosystems against timing violation attacks," in *Proc. IEEE Int. Symp. HOST*, Washington, DC, USA, 2015, pp. 72–77.

[5] A. Bal, S. Saha, S. Roy, and K. Chakraborty, "Revamping timing error resilience to tackle choke points at NTC systems," in *Proc. DATE*, Lausanne, Switzerland, 2017, pp. 1020–1025.

[6] K. Dunham, *Mobile Malware Attacks and Defense*. Burlington, MA, USA: Syngress, 2008.

[7] V. Southmayd. (Aug. 2016). *4 Patch Management Practices to Keep Your Clients Secure*. [Online]. Available: http://www.labtechsoftware.com/blog/4-patch-management-best-practices/

[8] T.-E. Wei *et al.*, "Android malware detection via a latent network behavior analysis," in *Proc. 11th IEEE Int. Conf. Trust Security Privacy Comput. Commun. (TrustCom)*, Liverpool, U.K., 2012, pp. 1251–1258.

[9] X. Jiang and Y. Zhou, *A Survey of Android Malware*. New York, NY, USA: Springer, 2013, pp. 3–20.

[10] J. Xin and R. Joseph, "Identifying and predicting timing-critical instructions to boost timing speculation," in *Proc. MICRO*, Porto Alegre, Brazil, 2011, pp. 128–139.

[11] R. Bertran *et al.*, "Voltage noise in multi-core processors: Empirical characterization and optimization opportunities," in *Proc. MICRO*, Cambridge, U.K., 2014, pp. 368–380.

[12] N. Ahmed, M. Tehranipoor, and V. Jayaram, "Supply voltage noise aware ATPG for transition delay faults," in *Proc. VTS*, Berkeley, CA, USA, 2007, pp. 179–186.

[13] A. M. Joshi, L. Eeckhout, L. K. John, and C. Isen, "Automated microprocessor stressmark generation," in *Proc. 14th Int. Conf. High Perform. Comput. Archit. (HPCA)*, Salt Lake City, UT, USA, Feb. 2008, pp. 229–239.

[14] Y. Kim and L. K. John, "Automated di/dt stressmark generation for microprocessor power delivery networks," in *Proc. ISLPED*, Fukuoka, Japan, 2011, pp. 253–258.

[15] W. Zhao and Y. Cao. (Jun. 2012). *Predictive Technology Model*. [Online]. Available: http://ptm.asu.edu/

[16] N. K. Choudhary *et al.*, "FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template," in *Proc. ISCA*, San Jose, CA, USA, 2011, pp. 11–22.

[17] *NanGate*. Accessed: Dec. 20, 2017. [Online]. Available: http://www.nangate.com/?page_id=2328

[18] H. Chen, S. Roy, and K. Chakraborty, "DARP: Dynamically adaptable resilient pipeline design in microprocessors," in *Proc. DATE*, Dresden, Germany, 2014, pp. 1–6.

[19] *Scikit-Learn, Machine Learning in Python*. Accessed: Dec. 20, 2017. [Online]. Available: https://scikit-learn.org

[20] F. Pagnotta and H. Amran, "Using data mining to predict secondary school student alcohol consumption," Dept. Comput. Sci., Univ. Camerino, Camerino, Italy, 2016.

[21] S.-M. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Trans. Comput.*, vol. 49, no. 9, pp. 967–970, Sep. 2000, doi: 10.1109/12.869328.

[22] J. Balasch, B. Gierlichs, and I. Verbauwhede, "An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs," in *Proc. Workshop FDTC*, Nara, Japan, 2011, pp. 105–114.

[23] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Proc. 16th Annu. Int. Conf. Theory Appl. Cryptograph. Tech. (EUROCRYPT)*, Konstanz, Germany, 1997, pp. 37–51.

[24] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Security Privacy*, San Jose, CA, USA, 2015, pp. 605–622.