Project Report
LSP-379

Side-Channel Assisted Real-Time Fuzzing for Embedded Systems (SCARFES): FY23 Cyber Security Line-Supported Program

K.W. McClintick
E. Binyamin
J.D. DeFrancesco
B. John
K.W. Ingols
B.R. Chetwynd
E.K. Shields

21 February 2024

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.

This report is the result of studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Massachusetts Institute of Technology
Lincoln Laboratory

Side-Channel Assisted Real-Time Fuzzing for Embedded Systems (SCARFES): FY23 Cyber Security Line-Supported Program

K.W. McClintick
E. Binyamin
J.D. DeFrancesco
B. John
K.W. Ingold
B.R. Chetwynd
E.K. Shields

Group 51

Project Report LSP-379

21 February 2024

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.

Lexington Massachusetts

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
21-02-2024

2. REPORT TYPE
Project Report

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE

On the Feasibility of Training an AI to Understand Programs: FY23 Cyber Security Line-Supported Program

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

K.W. McClintick, E. Binyamin, J.D. DeFrancesco, B. John, K.W. Ingols, B.R. Chetwynd, E.K. Shields

5d. PROJECT NUMBER
2231-4601

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02421-6426

8. PERFORMING ORGANIZATION REPORT NUMBER

LSP-379

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02421-6426

10. SPONSOR/MONITOR'S ACRONYM(S)
MIT LL

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

13. SUPPLEMENTARY NOTES

13. ABSTRACT

The U.S. government has a need to protect an increasing number of critical embedded systems (ES). Automated vulnerability assessments (VA) at scale are challenged by the vast diversity of ES.

The choice solution to ES VA is a combination of static and dynamic analysis enabled by a combination of source code, binaries, and development environments and/or digital twins used to develop the system. In cases where the development environment is not available to the government, as is often the case due to intellectual property concerns, for example, dynamic analysis can be enabled by rehosting the device in an emulator. The rehosting process today is manually intensive; In order for the embedded system to run, peripheral devices must be manually “stubbed out", a process that is dicult to automate due to the vast variety of peripherals and their implementations [1] Once emulated, this methodology allows for rapid coverage-based fuzzing, leveraging pairs of random program input and the code traces resulting from those inputs. In the absence of rehosting, some vulnerability assessment efforts have used blackbox fuzzing to harden systems of interest. This method carries significant disadvantages because the lack of instrumentation does not allow assessors introspection on the computation state that cannot be externally observed, including whether the system has crashed or what code paths were exercised by the system under test in response to input.

Although rehosting is quicker at discovering inputs for all traces through a CFG than random fuzzing and providing information useful for tracing execution to the bugs that cause crashes, it requires a large up-front labor investment which can be made larger in the case of uncommon ES architectures with few instrumentation tools available. Writing new machine models for rehosting software like Qemu or Renode can, anecdotally, take months of code development time. This upfront investment is prohibitive because the government needs to conduct assessments on far more system types than there exist reverse engineering experts to rehost them.

A middle-ground methodology that would allow some level of instrumentation for dynamic analysis without the upfront cost of rehosting would allow for initial investigations of target systems to triage a combination of the most critical and least resilient systems, which could be used to triage a large number of ES to better decide which systems to invest in rehosting. We investigate the feasibility of such a methodology, which we call Side-Channel Assisted Real-Time Fuzzing for Embedded Systems (SCARFES), on an STM32 microcontroller unit (MCU) running a toy C program. We show that SCARFES requires little more investment to set up than blackbox fuzzing, but demonstrates initial promise in the ability to leverage electromagnetic side channels to discriminate instruction traces in dynamic analysis.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT UNCLASSIFIED	**b. ABSTRACT** UNCLASSIFIED	**c. THIS PAGE** UNCLASSIFIED	None	52	**19b. TELEPHONE NUMBER** *(include area code)*

This page intentionally left blank.

(U) ABSTRACT

(U) The U.S. government has a need to protect an increasing number of critical embedded systems (ES). Automated vulnerability assessments (VA) at scale are challenged by the vast diversity of ES.

(U) The choice solution to ES VA is a combination of static and dynamic analysis enabled by a combination of source code, binaries, and development environments and/or digital twins used to develop the system. In cases where the development environment is not available to the government, as is often the case due to intellectual property concerns, for example, dynamic analysis can be enabled by rehosting the device in an emulator. The rehosting process today is manually intensive; In order for the embedded system to run, peripheral devices must be manually "stubbed out", a process that is difficult to automate due to the vast variety of peripherals and their implementations [1] Once emulated, this methodology allows for rapid coverage-based fuzzing, leveraging pairs of random program input and the code traces resulting from those inputs. In the absence of rehosting, some vulnerability assessment efforts have used blackbox fuzzing to harden systems of interest. This method carries significant disadvantages because the lack of instrumentation does not allow assessors introspection on the computation state that cannot be externally observed, including whether the system has crashed or what code paths were exercised by the system under test in response to input.

(U) Although rehosting is quicker at discovering inputs for all traces through a CFG than random fuzzing and providing information useful for tracing execution to the bugs that cause crashes, it requires a large up-front labor investment which can be made larger in the case of uncommon ES architectures with few instrumentation tools available. Writing new machine models for rehosting software like Qemu or Renode can, anecdotally, take months of code development time. This upfront investment is prohibitive because the government needs to conduct assessments on far more system types than there exist reverse engineering experts to rehost them.

(U) A middle-ground methodology that would allow some level of instrumentation for dynamic analysis without the upfront cost of rehosting would allow for initial investigations of target systems to triage a combination of the most critical and least resilient systems, which could be used to triage a large number of ES to better decide which systems to invest in rehosting. We investigate the feasibility of such a methodology, which we call Side-Channel Assisted Real-Time Fuzzing for Embedded Systems (SCARFES), on an STM32 microcontroller unit (MCU) running a toy C program. We show that SCARFES requires little more investment to set up than blackbox fuzzing, but demonstrates initial promise in the ability to leverage electromagnetic side channels to discriminate instruction traces in dynamic analysis.

This page intentionally left blank.

(U) TABLE OF CONTENTS

This page intentionally left blank.

(U) LIST OF FIGURES

(U) LIST OF FIGURES

(Continued)

1. (U) INTRODUCTION

(U) [1]The Department of Defense (DoD) utilizes a large number of embedded systems (ES), including avionics, weapons systems, and RF equipment. Many of these systems are legacy systems for which modern security analysis techniques have not been applied. A key tool for dynamic analysis is a fuzzer [2], which permutes inputs in order to stress the system's ability to handle malformed and malicious interactions, thus detecting bugs and vulnerabilities. However, effective fuzzing requires an understanding of which inputs caused the system to malfunction, which usually requires instrumentation of the system. This is more tractable for traditional desktop computing software and hardware, which can sometimes be easily hosted on similar systems, but not for embedded systems, which often utilize different architectures than enterprise systems and utilize peripherals that need to be modeled. "Rehosting" embedded systems into a desktop environment, in the face of vendor IP restrictions and non-standard or legacy architectures, still requires significant upfront manual efforts to program new virtual machines, which is labor-limited. There is a need to perform vulnerability assessments (VA) of embedded systems with little upfront manual effort, such that labor may be more appropriately allocated to high-vulnerability systems (Fig. 1).

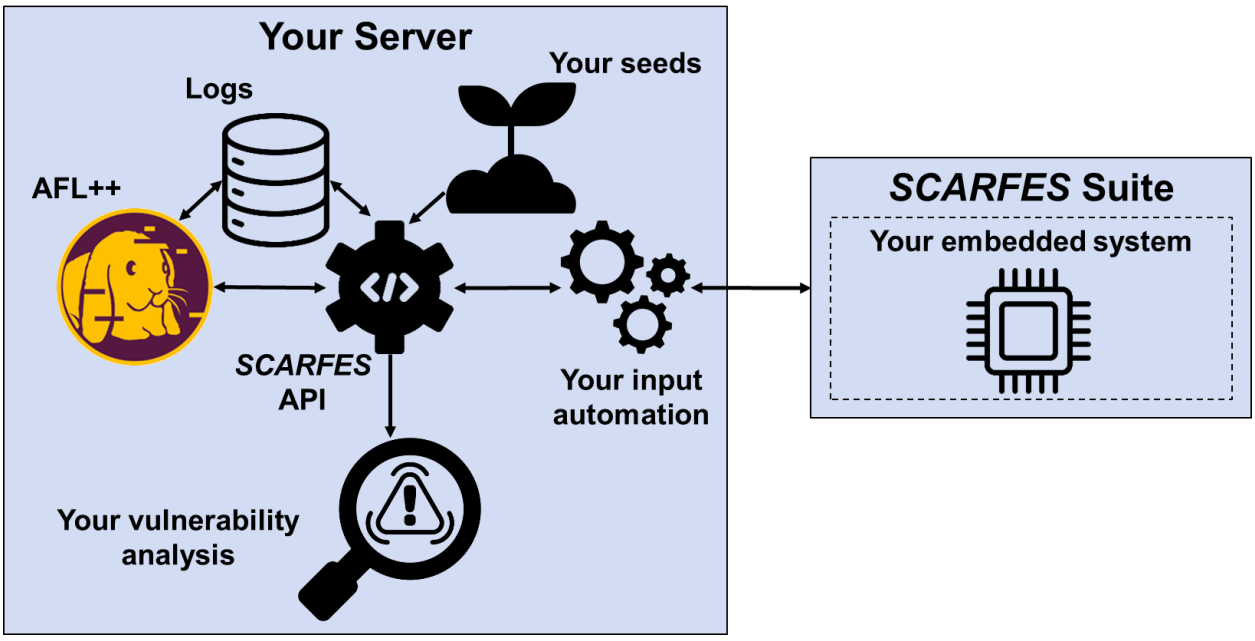

Figure 1. (U) OV-1 of how SCARFES could be used to assist in VA tasks.

[1]This work serves as the end-of-year report for SCARFES 2231-4601 TI15-2301, whose PI is Kyle McClintick, due October 31, 2023, in accordance with the Line and Technical Investments Guidance Memo, section 4.3. This document is to be found in the Cybersecurity line's SharePoint folder, reporting to kimberly.pitko@ll.mit.edu, dbigelow@ll.mit.edu, and Dennis.Ross@ll.mit.edu.

(U) The Side-Channel Assisted Real-Time Fuzzing for Embedded Systems (SCARFES) project seeks to enable a side-channel-enabled coverage mode for fuzzing through the use of unintended electromagnetic emissions. Other research related to this topic has been done to fingerprint (without fuzzing) the sample program ran from the Arduino Integrated Development Environment (IDE) [3] via radio frequency (RF) side-channels, fuzz by fingerprinting the RF side-channels of basic blocks and basic block transitions of a custom program on a STM32F417 Microcontroller Unit (MCU) [4], or fuzz by computing the side-channel cost (e.g., execution time or response size) difference between emulated execution pairs [5,6]. Finally, unsupervised methodologies for differential power analysis (DPA) has been investigated [7].

(U) The SCARFES project, funded through the Cybersecurity Line, is developing a solution to provide a side-channel vulnerability assessment testbed and fuzzing software for ESs that cannot easily be rehosted due to IP restrictions or because rehosting a new virtual machine due to a non-standard or legacy architecture is an undesired labor cost. This is accomplished by extracting information from unintended electromagnetic emissions leaked during the execution of software on an ES. By leveraging side-channels and unsupervised algorithms, we infer software instructions and traces. By doing so, we determine when a software execution has discovered new code coverage, such as a conditional jump between memory not seen in previous software executions. Over time, one could identify a common conditional branch pattern of instruction IDs, and therefore detect branch points before both paths have been exercised. We leave this novel method to future work. Coverage is the fuzzing primitive that is measured by CFG execution paths in smart fuzzers, and is used to guide software input mutations to maximize the speed at which the CFG is explored for the purpose of finding vulnerabilities in the program.

(U) The project's use of unsupervised learning is our primary contribution to the state of the art. Supervised approaches require a twin system for which instruction execution can be tightly controlled for fingerprinting, while our unsupervised approach allows characterization of systems without modifying their software or firmware. Fingerprinting measurements require the extraction of ES memory to be flashed back later, as well as flashing a set of programs with accurate time-domain knowledge of which instruction is being executed during each clock cycle. This methodology is unreliable because the physical attributes of silicon are not robust to physical changes [8]. That is to say, fingerprints are not robust to small environmental changes such as temperature, pressure, humidity, and antenna positioning. Our methodology is robust to small changes over time, as we compare traces to their nearest side-channel neighbor. By leveraging side-channels to perform fuzzing without instrumentation or rehosting, the SCARFES project may help DoD software security experts efficiently test a diversity of embedded systems used in operation for initial vulnerability assessment triage.

(U) We contribute in the following ways to the state of the art for RF side-channel fuzzing of ESs.

- A novel unsupervised approach for discovering coverage via RF side-channels for use in a partially observable CFG fuzzer

- An American Fuzzy Lop (AFL++) [9] UNIX socket inter-process communication (IPC) proxy, which allows for custom coverage primitive communication with AFL++; coverage primitives may be inferred via SCARFES or any other methodology

(U) The rest of this manuscript is organized as follows.

- Section II discusses the assumptions and implementation of our coverage inference, leveraging unsupervised learning.
- Section III describes our target ES, target program, communications between the server and the ES, and how RF side-channels are measured.
- Section IV presents example RF side-channel measurements, metrics pertaining to coverage inference, and fuzzing curves.
- Section V discusses the impact of this work and reviews open challenges.

This page intentionally left blank.

2. (U) METHODOLOGY

(U) Our non-invasive (no soldering or decapping) methodology[2] is formulated under the following assumptions.

- Changing currents during program execution should induce electromagnetic fields, such that printed circuit board (PCB) traces, surface-mounted device (SMD) components, and integrated circuits (ICs) can be thought of as antennas [10].
- Program execution will display different electromagnetic signatures as a function of which assembly instruction was executed [11].
- In synchronous sequential circuits, a significant part of the RF side-channel leakage is attributed to the clock distribution network [12].
- To precisely read side-channel emissions with a highly directive and near-field receiver off an ES, a small loop antennae is required. Smaller antennas are optimized to receive higher frequencies, and state-of-the-art works attribute instruction-correlated RF signatures to higher-than-clock frequencies, often clock harmonics [3].

(U) With these in mind, we argue that RF side-channel time series signals for fuzzing should be interpreted in 2D data structures, organized and triggered by the clock cycle on the first dimension. For example, a device with a 100 MHz clock whose side-channels are sampled by an oscilloscope with a 1 GHz sample rate should organize its captures into 10-sample clock cycles. We hypothesize that, if enough clock cycles can be accurately clustered by instruction ID, we may be able to infer coverage branching. One may relate this task to that of the harder RF side-channel disassembly problem [13–15] or the inference of the identity and order of assembly instructions executed by an ES.

(U) Since our task is to search for higher-than-clock frequency novelty in RF side-channel signals for an unknown number of instructions, we argue for the use of anomaly detection algorithms in place of clustering algorithms (i.e., K-means [16]), since clustering has strong assumptions of cluster number or data distributions. Further, we argue for the use of non-parametric anomaly detection algorithms, since parametric approaches [17, 18] require prohibitively lengthy training for fuzzing applications. Fuzzing requires scaled inference for experiments that last weeks or even months, so the cost of inferring the coverage of an execution must be low, specifically on the order of milliseconds or less. With this goal, we also aim to minimize the amount of averaging used in our methodology, which other side-channel approaches tend to use liberally, on the order of thousands of executions per input. We do not consider isolation forest (IF) [19] because each clock cycle group will begin with just one training data point. With only one training data point, IF will always only need one split to separate training from inference data such that it will either always infer no anomalies or always infer that an anomaly has been detected.

(U) Given these goals and constraints, we consider three algorithms: the one-class support vector machine (OCSVM) [20]; hypothesis tests, specifically Kolmogorov-Smirnov (KS) two-sample

[2] code available: https://llcad-github.llan.ll.mit.edu/SCARFES/scarfeprobed, includes python virtual environment

tests [21], and unsupervised outlier detection using empirical cumulative distribution functions (ECOD) [22]. OCSVM uses the kernel trick to solve non-linear problems, where outliers lay beyond the radius of a hyper-sphere and in-class data lay within. It is known to be sensitive to outliers, making it more appropriate for novelty detection when the training data are not contaminated with outliers. This is not guaranteed, which is to say that errors can compound if an outlier is falsely allowed into the training data for a class. KS and ECOD tests utilize the cumulative distribution function (CDF) of data. Specifically, the KS test determines a threshold statistic from the maximum CDF distance between training and testing sets. Meanwhile, ECOD first estimates the underlying distribution of the input data in a nonparametric fashion by computing the CDF per dimension of the data. ECOD then uses these empirical distributions to estimate tail probabilities per dimension for each data point. Finally, ECOD computes an outlier score of each data point by aggregating estimated tail probabilities across dimensions.

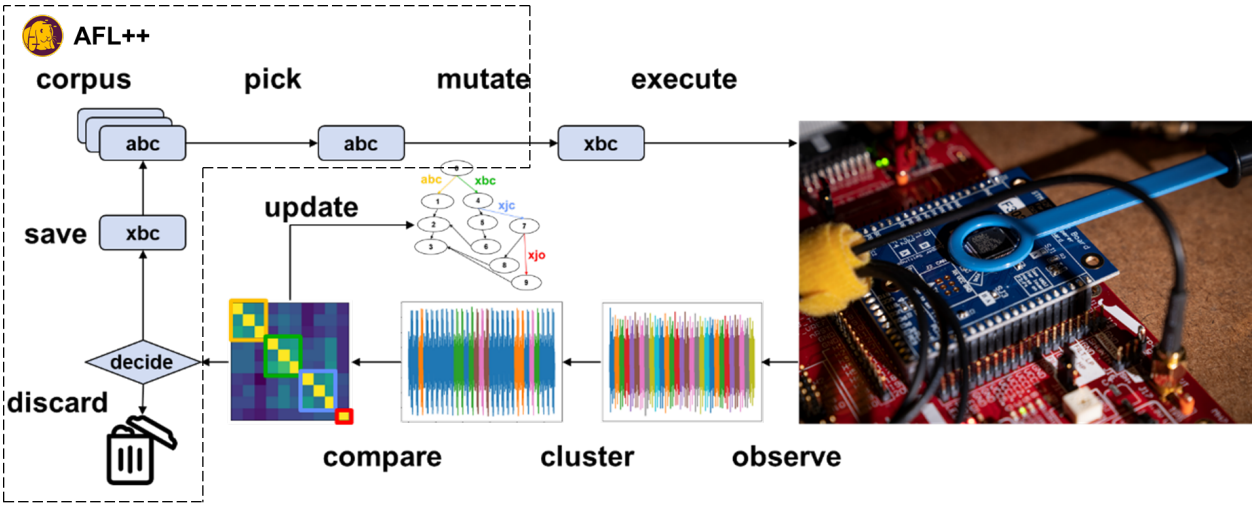

Figure 2. (U) A state-of-the-art fuzzing cycle, modified for our partially observable CFG side-channel approach. We utilize AFL++ for its corpus management and mutation engine, but interface with a UNIX socket server to our python scripts, which infer coverage from measurements exported by the oscilloscope.

(U) We present our algorithm (Algorithm 1, Fig. 2), which first segments triggered side-channel oscilloscope EM captures into 2D structures, where each row represents one clock cycle. Then, each clock cycle is compared to each existing side-channel class via a non-parametric anomaly detection algorithm (i.e., OCSVM or ECOD). For the first clock cycle of the first capture analyzed in this way, the first side-channel class is created. If the candidate clock cycle is an outlier for all existing side-channel classes, then a new side-channel class is created and populated with it. Otherwise, the clock cycle is added to the most similar side-channel class as an in-class example, and the anomaly detector for that class is fine-tuned. After all clock cycles of the current capture are classified in this way, we will have inferred a side-channel class state transition sequence, represented by a list of integers.

This sequence is added to a memory structure, and if a transition is seen that does not exist in memory (e.g., a class 7 clock cycle followed by a class 9 clock cycle), then that capture is said to contain novel coverage that our fuzzer should reward by adding the program input to the corpus.

(U) AFL++ is intended to interpret coverage in the form of basic blocks, or sequences of instructions that never branch. With proper instrumentation, all blocks are known before fuzzing begins. In our experiment, blocks would have to be discovered in a constantly shifting control flow graph (CFG), which AFL++ is not optimized for. This metric is given in the AFL++ status screen as "stability", or the percentage of inputs that give the same coverage each time they're executed. That is to say, the first program execution's returned coverage would just be one basic block, $MC(0) = [0]$, and, if the next execution's side-channel class sequence deviates from the previous execution, would create a new basic block changing $MC(0) = [0 \rightarrow 1]$ and returning $MC(1) = [0 \rightarrow 2]$. For 100% stability, we provide a simple methodology (Algorithm 1) to communicate two values to AFL++: the integer list of clock cycle class predictions for the current execution, and a Boolean communicating if those predictions contain any novel side-channel clock class edges.

(U) This stable, clock cycle edge inference methodology may be extended to infer EM side-channel state blocks, which are analogous, but not necessarily estimates of, CFG basic blocks. Basic blocks are the unconditional sequences of instructions of a program. By communicating to AFL++ which side-channel state block edges are novel, rather than that any clock cycle edges are novel, better search optimization may be performed at the cost of stability.

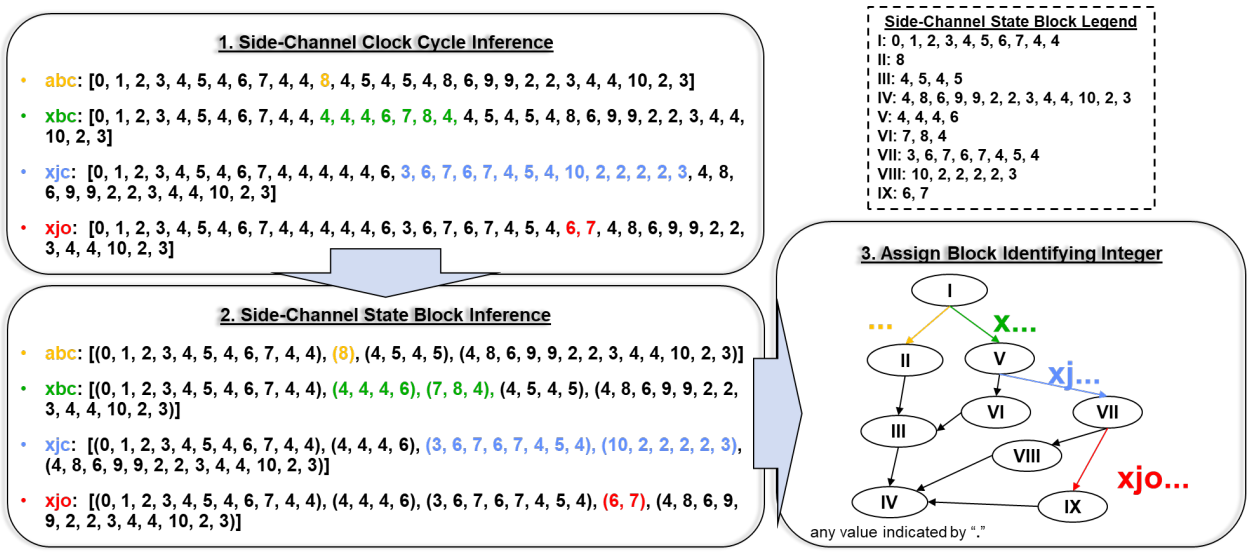

Figure 3. (U) An example of our low-stability, high-information coverage inference methodology.

Algorithm 1 (U) SCARFES Program Clustering Algorithm

Given: Collect X_{mn}, clock cycles m, samples/clock cycle n, number of classes C, clock data memory $MX_{C\times 0\times n}$, clock class memory $MC_{P\times m}$, current capture index $p = 1, ..., P$, anomaly detector $f_C(X(a), X(b)) \in [0, 1]$
$Cov \leftarrow False$
Reshape $X_{m\times n} \leftarrow X_{mn}$
$C \leftarrow 0$
for $i = 1, ..., m$ **do**
    if C **is** 0 **then**
        $MX \leftarrow [M, X(i)]$
        $MC(p, i) \leftarrow C$
        $C \leftarrow C + 1$
    else
        $MinDist = \infty$
        $MinDistClass = \varnothing$
        for $j = 1, ..., C$ **do**
            for $k = 1, ...m_C$ **do**
                if $f(MX(j, k), X(i) \leq 0.5$ **and** $< MinDist$ **then**
                $MinDist \leftarrow f(MX(j, k), X(i)$
                $MinDistClass \leftarrow j$
                end if
            end for
        end for
        if $MinDistClass$ **is** $\varnothing$ **then**
            $MX \leftarrow [MX, X(i)]$
            $C \leftarrow C + 1$
            $MC(p, i) \leftarrow C$
            Train $f_C()$ with $X(i)$
        else
            $MX \leftarrow [MX(MinDistClass), X(i)]$
            $MC(p, i) \leftarrow MinDistClass$
            Train $f_{MinDistClass}()$ with $X(i)$
        end if
    end if
end for
if any pair $MC(p)$ **not in** $MC($**not** $p)$ **then**
    $Cov \leftarrow True$
end if
return $MC(p)$, Cov

(U) To visualize this process, consider Figure 3, which performs inference on side-channel collects from Section 4. The details of this collect may be left for later. For now, know that the program that was executed parses arrays of characters with three conditional branches. Should the input start with 'x', the green branch will be taken instead of orange. Should the second character also be 'j', the blue branch will be taken. Should the third be 'o', the red branch will be taken. All traces end with basic block IV, and start with I. The first stage of inference, "side-channel clock cycle inference" corresponds to Algorithm 1, assigning an integer class value to clock cycles that are anomalies, creating classes 1 through 10. After each input is executed, these integer sequences are broken into their longest common sequences. This is shown by box 2, "side-channel state block inference", which displays the results of this branch detection after "xjo" has been executed for the set of four executions abc, xbc, xjc, and xjo. The results of assigning yet another integer identifier to these common sequences is shown in box 3, "assign block identifying integer", which a legend pointing back to box 2 provided above it. This is what we communicate to AFL++ in our low-stability and high-information methodology, which would infer the following side-channel state block sequences: $MC(0)$ =I→II→III→IV for "abc", $MC(1)$ =I→V→VI→III→IV for "xbc", $MC(2)$ =I→V→VII→VIII→IV for "xjc" and $MC(3)$ =I→V→VII→IX→IV for "xjo".

(U) This methodology contributes to the state-of-the-art by outlining a framework for using off-the-shelf fuzzers like AFL++ to optimize input mutation and corpus management by formatting side-channel inference rather than a harnessed executable. We do this in the form of clock cycle and state block edge traces. In this way, AFL++ discovers an unknown number of blocks through anomaly detection rather than exploring a known number as provided by a CFG. We note that this methodology is an example of the classic "hatling problem", and it is up to the operator to decide when to stop fuzzing, as there is no way to know if all blocks have been discovered.

This page intentionally left blank.

3. (U) EXPERIMENTAL SETUP

(U) We demonstrate the feasibility of our methodology and experimental suite (Fig. 4) on the STM32 Nucleo-144 development board with STM32F413ZH MCU and a 96 MHz clock because of the complexity and popularity of STM32F413xG/H devices, which are based on the high-performance Advanced RISC Machine (ARM) Cortex-M4 32-bit Reduced Instruction Set Computer (RISC) core. Cortex-M cores are often used in power management, I/O, system, touch screen, smart battery, and sensors controllers. This choice of target complicates side-channel analysis because it utilizes a five-stage pipeline, meaning each clock cycle is actually processing up to five instructions at a time, and random pipeline refills extend the number of clock cycles required to perform instructions by a stochastic amount. The target also performs out-of-order assembly instruction executions, meaning instruction traces and actual traces will not match due to order optimization. The target further performs speculative execution, meaning it will pre-load the pipeline with instructions that may or may not be correctly predicted, necessitating pipeline flushes in the case of incorrect speculations. One might leverage this to identify branch instructions, but we leave this to future work. Finally, the MCU regulates voltage to a constant value, smoothing the variations in power consumption.

Figure 4. (U) A photograph of our experimental suite in the HSATp lab, B-317A.

(U) Although SCARFES and the broader fuzzing concept is not limited to data parsers (i.e., libxml, libpdf, libyaml), we utilize them in this demonstration of feasibility because they are often studied software targets for fuzzers. This is because data parsers have exploitable vulnerabilities, process attacker-controlled input, and use complex logic that can be buggy. Here at the lab, we perform many electrical and computer engineering estimation and classification tasks that involve the use of ESs to parse series of sampled sensor data in the presence of noise or interference (e.g., radar, RF communications, optical communications, audio processing). Consequently, we designed a toy data parser program, in C, whose execution path is determined directly by program input, and not by any sort of program state. We diversify the instructions executed by each basic block to see how unique that will make the RF side-channel leakage. The relevant portion of the main method is as follows.

```
while (1)
{
   getSerialData (rxBuffer); // Rx mutated fuzzer data from AFL++ over UART
   if (!isEmpty (rxBuffer)) {
      HAL_GPIO_WritePin(trigger_GPIO_Port, trigger_Pin, 0); // trigger low
        // check if first character is right
        if( rxBuffer[0] == 'x' ) {
        // check if second character is right
            for(int i = 0; i < 10; ++i) {
                rxBuffer[0]++;
            }
            if( rxBuffer[1] == 'j' ) {
                // do some math. If 'o', /= 0 (floating point exception)
               rxBuffer[0] /= (rxBuffer[1] - rxBuffer[2] + 5);
            }
         }
      HAL_GPIO_WritePin(trigger_GPIO_Port, trigger_Pin, 1); // trigger high
   }
   memset(rxBuffer, 0x00, MAX_BUFF_SIZE); // reset buffer value
}
```

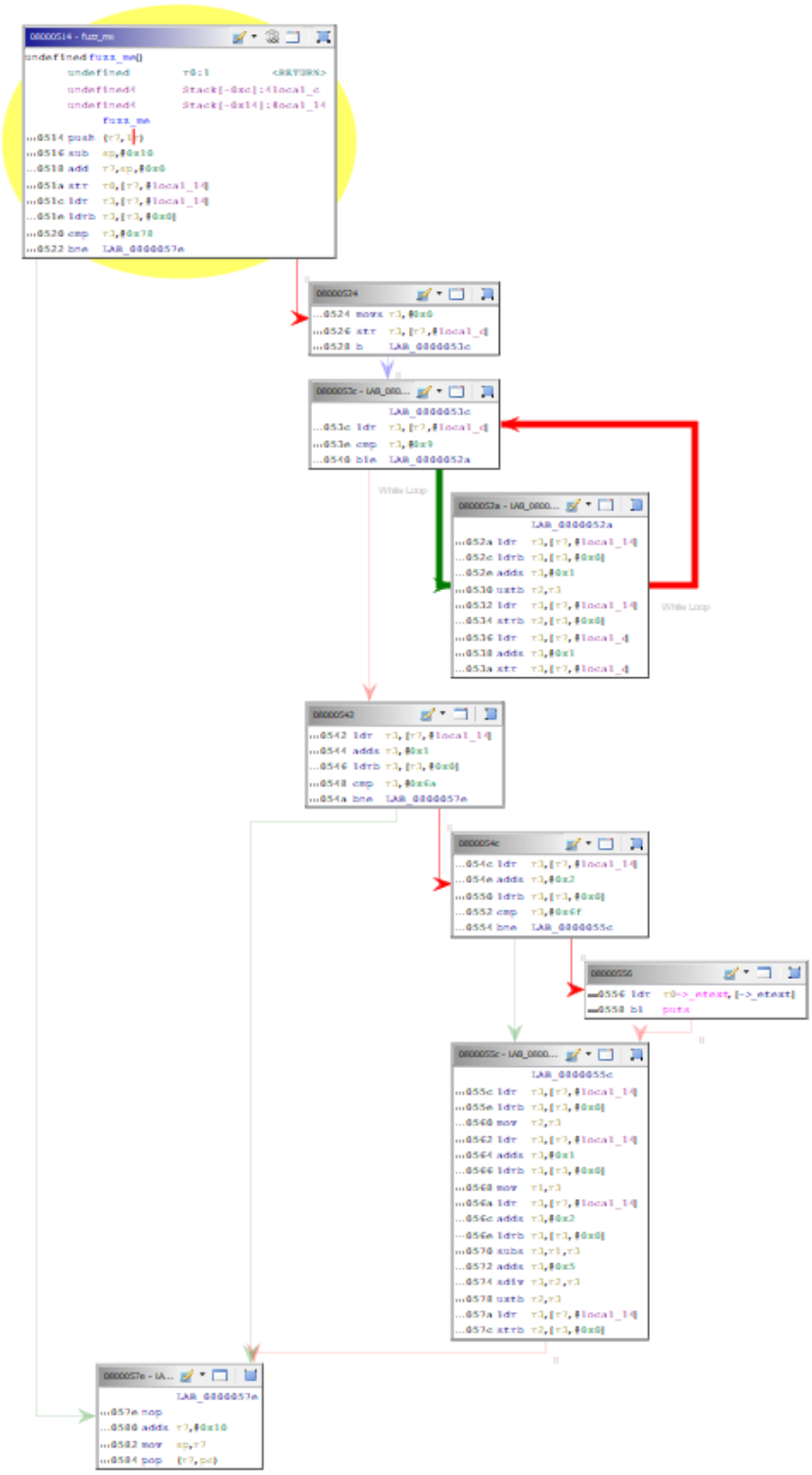

Figure 5. (U) A screenshot of the toy C program's CFG, as analyzed by Ghidra as an elf file with an ARM:LE:32:v8 (1.106) language.

(U) This program continuously queries for universal asynchronous receiver-transmitter (UART) data from the server computer, resetting the value in memory to zeros before each query. If UART data are received, a GPIO trigger will go from high to low, signaling the start of instruction execution for which we are interested in fuzzing. This signal would not be present in real programs, but we give it to ourselves to hasten development of the project. The triggering of side-channel analysis is a topic for future research, but we anticipate that the start and stop of program execution could in some cases be detected by high SNR side-channel signals such as on-chip peripheral communications (e.g., cryptography rounds [12]), off-chip peripheral communications (e.g., UART), and "idling" fingerprinting (i.e., what the while loop side-channel looks like when no input is received). If the first character of the UART data is not an "x", the program signals its end with a GPIO trigger going from low to high. Otherwise, a for loop will increment the value of the UART data. Then, if the second character of the UART data is "j", the algorithmic logic unit (ALU) will perform some instructions. If the third character of the UART data is "o", then the sdiv instruction will result in a floating point exception (FPE). If this is the case, the sdiv instruction crashes the program, triggering a reboot and skipping the trigger high command (trigger stays low), causing the program to idle until the next UART transmission is received. We expect the execution of this program to follow a CFG comprised of three branches, where UART inputs starting with "x", "xj", "xjo", and inputs not starting with "x" will produce one of four different execution paths through basic blocks. We utilize the 7049GP-TRT SuperWorkstation as our server computer to minimize the time elapsed for inference computations.

(U) Ghidra is a reverse engineering tool that we use to analyze our elf file (Fig. 5) to obtain instruction a CFG. The CFG tree displays a starting block, three conditional branches, an unconditional for loop block, and a function exit block. The diagram tells us that our program has not been optimized to behave differently than we designed it to. Unfortunately, although Ghidra supplies an instruction trace for each input given to the program and online documentation[3] supplies the number of clock cycles required for each instruction, these values vary randomly with "P", the pipeline refill value, making it difficult to obtain the ground truth of side-channel captures. We define ground truth to mean knowledge of which instruction was being executed during each clock cycle with total certainty.

(U) We build our source code on the server into a .hex file using STM32CubeIDE, flashing over a Serial Wire Debug (SWD) connection (Fig. 6) using st-link. We also send mutated inputs from AFL++ over a UART using pyserial, receive general-purpose input/output (GPIO) triggers over the Tektronix MSO64B Series oscilloscope at 25 GS/s, and export measurements to the server using pyvisa. EM measurements collected by the Langer ICR HV500-75 near-field microprobe (200 kHz–1 GHz) are first filtered and amplified by a Langer 706 bias tree. Our target device is shielded and absorbed (20 MHz–0 GHz) by a Ramsey STE3600 RF Test Enclosure.

[3] https://developer.arm.com/documentation/ddi0439/b/CHDDIGAC

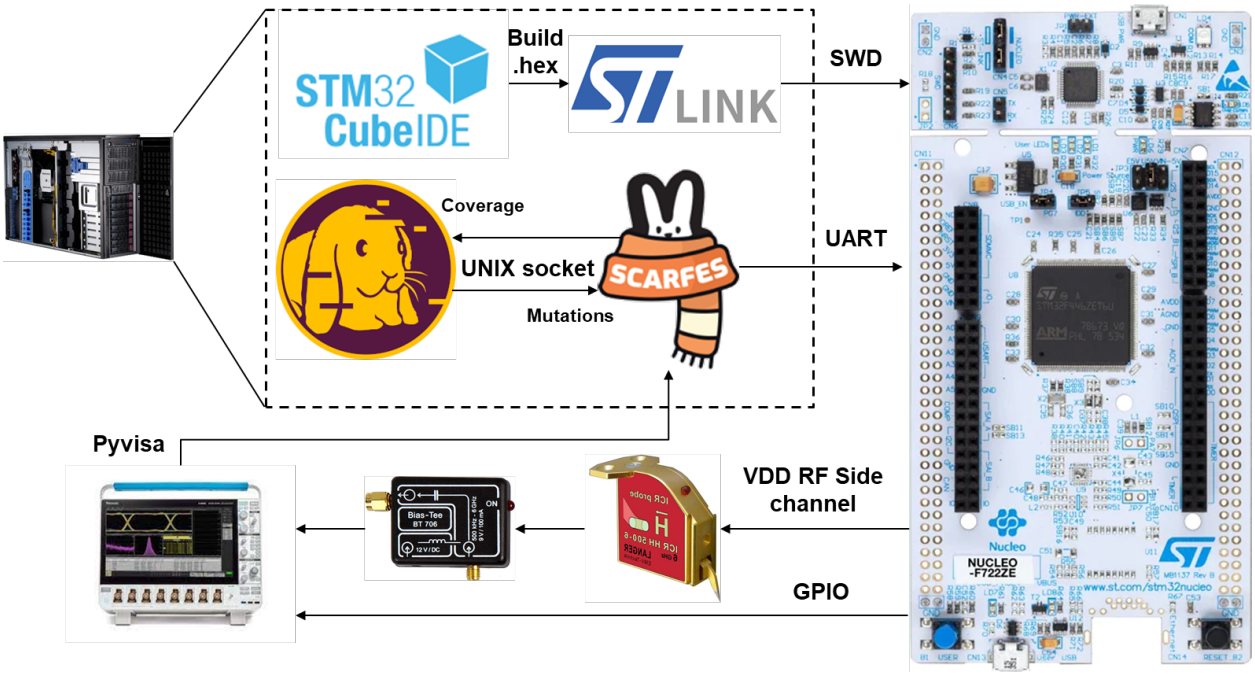

Figure 6. (U) An overview of our experimental setup, highlighting connections between devices.

(U) Side-channel analysis of MCU power is traditionally performed via current probes on resistance stable shunts [23] of voltage drain (VDD) pins, typically requiring soldering to put the shunt in series on a breadboard. To sense power non-invasively, we utilize a near-field directional microloop antenna probe designed to isolate and sense power from individual PCB traces and MCU pins at a distance of less than 1 mm. Knowing the typical current loop of SMD capacitors (Fig. 7), we measure side-channel RF leakage from three orthogonal loop orientations at various distances and polarities (i.e., x, y, z) of VDD pins and SMD components connected to those pins. We found the best signals correlated to code execution by measuring the high voltage side of SMD VDD capacitors, more so than SMD resistors and MCU pins on VDD lines. The best orientation is a vertically aligned orientation, in-line with the current loops longest path, placed as close as possible to the capacitor, without touching it. We make the use of positioners to do this.

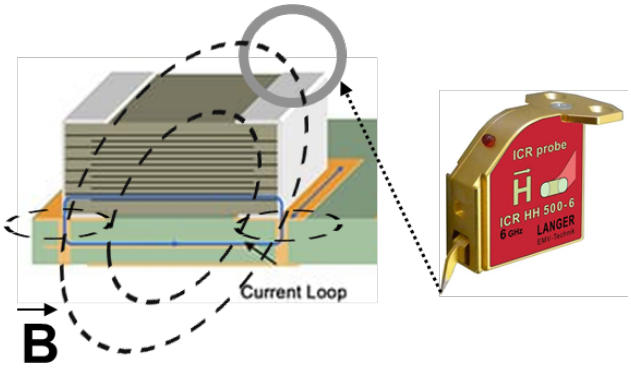

Figure 7. (U) By testing various EM probe locations and orientations, we conclude SMD VDD capacitors such as that pictured emit the strongest side-channel frequencies that are correlated to code execution.

(U) We utilize a UNiplexed Information Computing System (UNIX) domain socket for IPC between AFL++ and our python scripts. We encode UCS (Unicode) Transformation Format (UTF)-8 coverage information in the format "COV:0,1,2:0\r\r\n" as a byte buffer, which in this example would communicate clock cycle IDs 0, 1, and 2 were executed by the previous program execution, which is given input received from the socket server in the form of a 1024-byte buffer. The example also communicates a status flag in the integer set ranging from 0 to 255. A "0" would correspond to no crash or hang. Error status flags are application specific, but a "1" would usually indicate a crash. Other non-zero values can be used to communicate different types of crashes or hangs. AFL++ uses this status flag to organize logs. We leave inference of this status flag from side-channel data as future work, but we hypothesize that a hang could be detected if a program execution end trigger is not detected, and a crash could be inferred to have occurred if an "idle/reboot" side-channel pattern is detected, since bare-metal ESs will reboot when a crash occurs.

4. (U) RESULTS

(U) Figures 8–11 displays example side-channel collects from the oscilloscope for one input from each of the four possible code execution paths for our toy program (see Fig. 5.) Notice that the GPIO trigger (green) stays low for longer when more instructions are executed. Also notice the annotated 96 MHz clock signal in the Channel 3 spectrum view (white). We observe a highly periodic EM signal (red) when GPIO trigger is high, which is when the NUCLEO is "idling". Frequencies below the clock correlate strongly to execution duration, which is known to not correlate to code execution path due to structures like loops. That is to say, just because two executions take different amounts of time does not mean a different conditional branch was taken, the input may just be longer, for example. Additionally, frequencies near 20 MHz and below are not blocked by our enclosure.

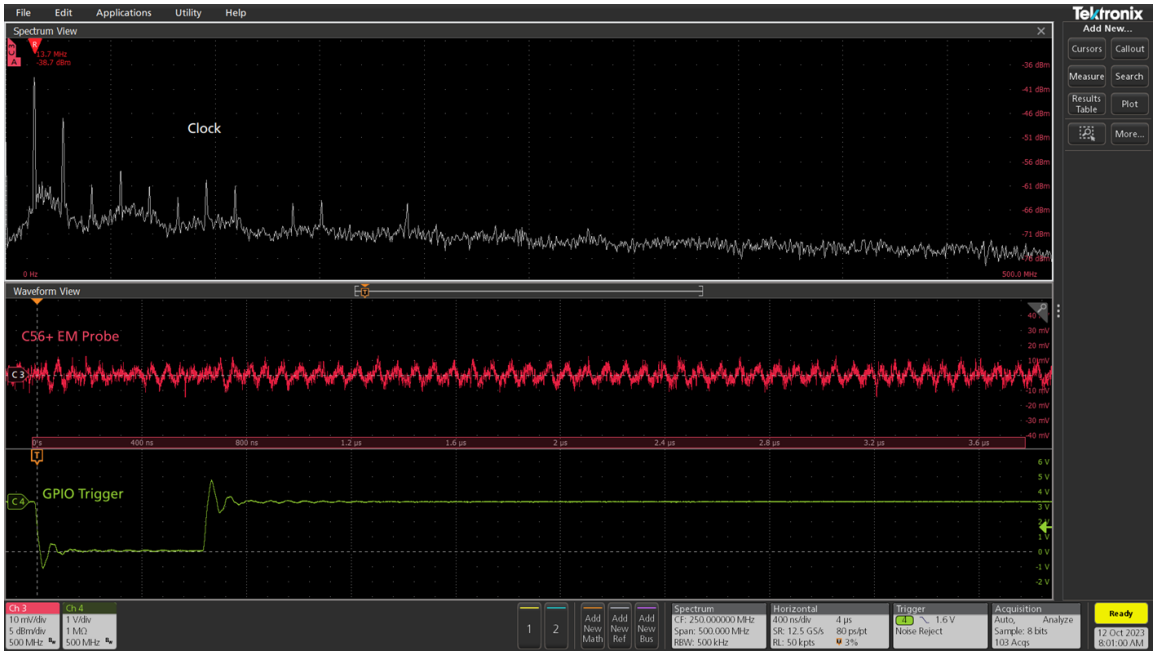

Figure 8. (U) Side-channel oscilloscope screen capture for "abc" program input, representing an example of conditional branch one of the CFG not taken. Spectrum view is for Channel 3. The 96 MHz clock frequency is annotated.

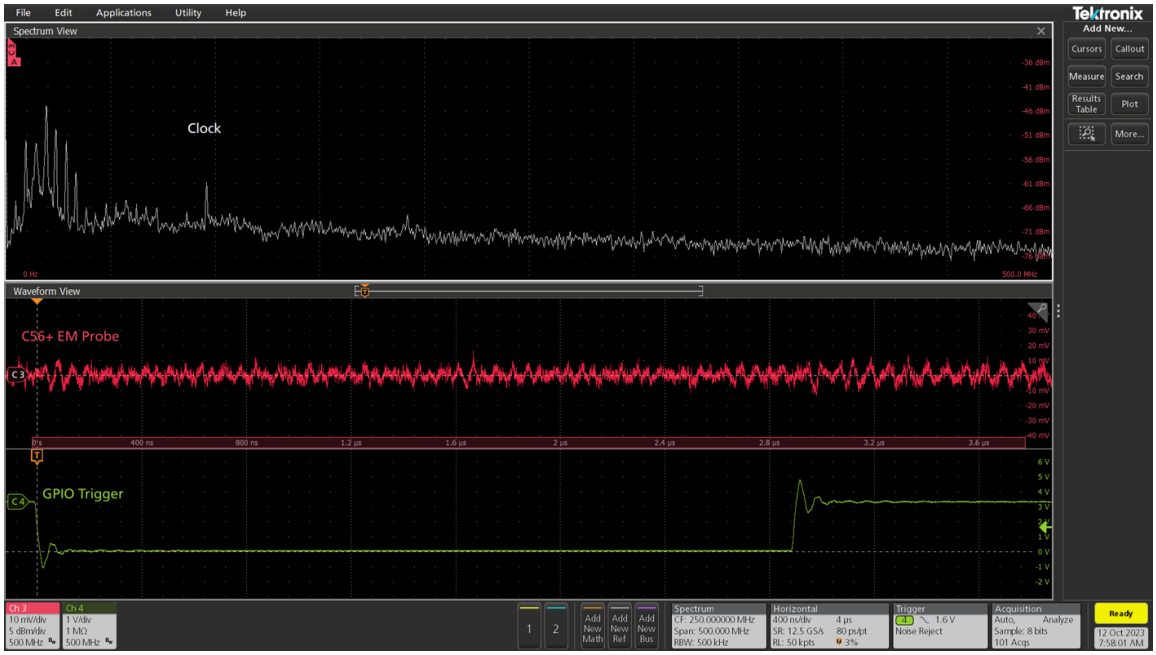

Figure 9. (U) Side-channel oscilloscope screen capture for "xbc" program input, representing an example of conditional branch one of the CFG taken.

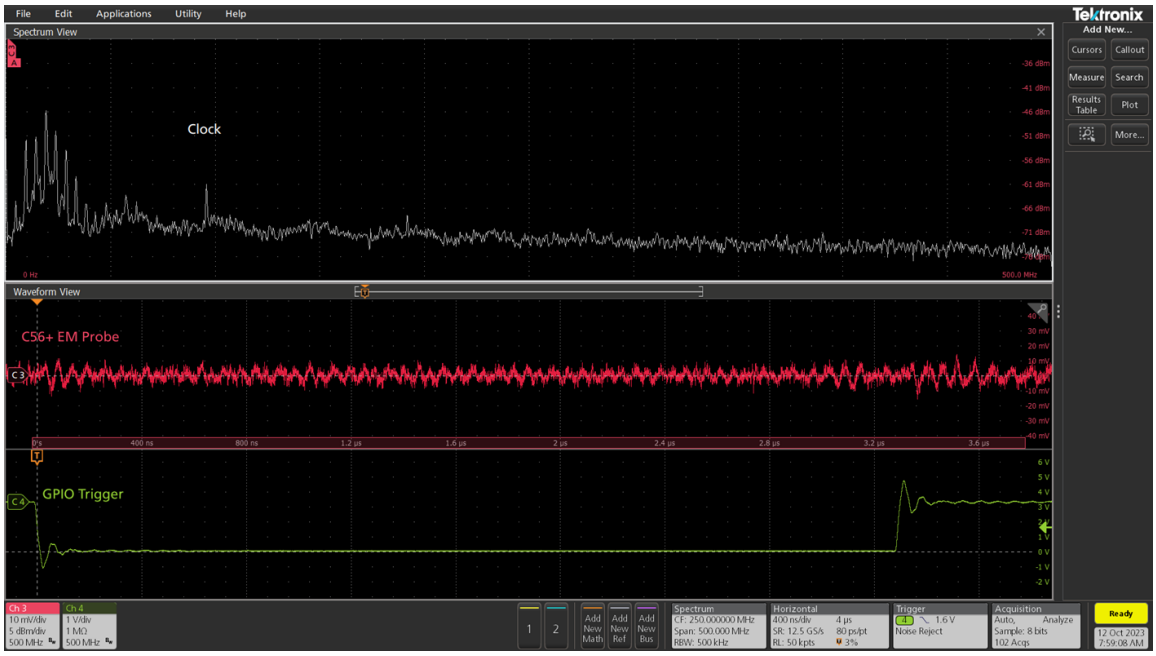

Figure 10. (U) Side-channel oscilloscope screen capture for "xjc" program input, representing an example of conditional branch one and two of the CFG taken.

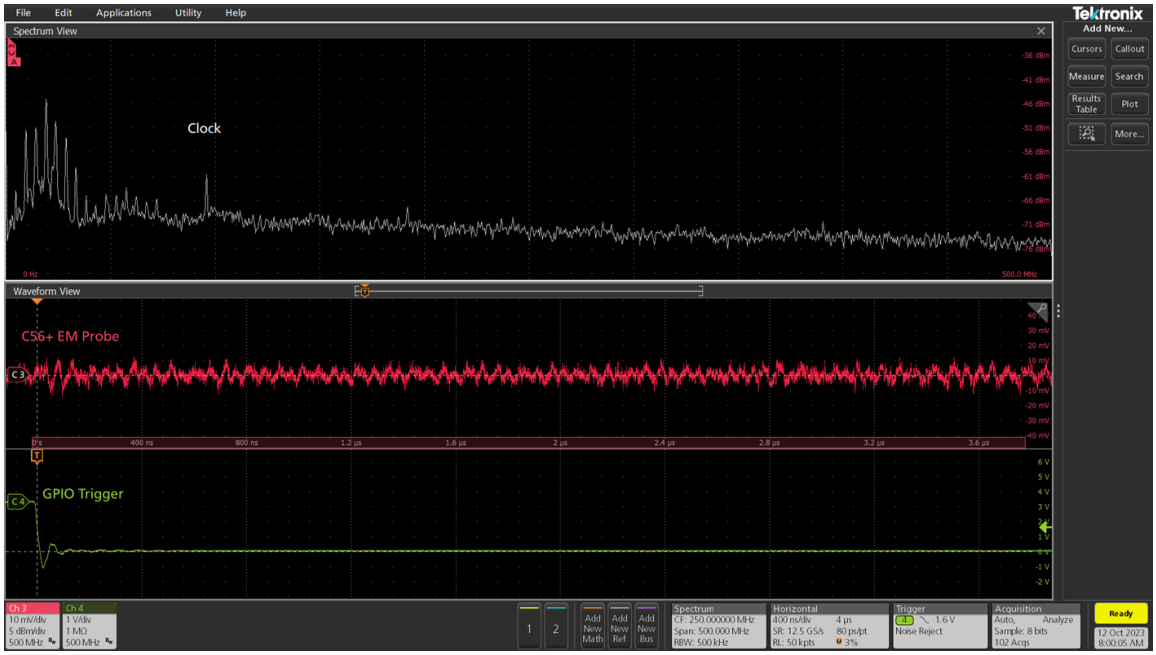

Figure 11. (U) Side-channel oscilloscope screen capture for "xjo" program input, representing an example of conditional branch one, two, and three of the CFG taken.

(U) Consequently, we are most interested in frequencies above the 96 MHz clock for the purpose of determining which instructions were executed and when to determine code execution path. Utilizing our methodology outlined in Section 2, we obtain small-scale inference results for the following program inputs: abc, zyx, xbc, xyz, xjc, xjz, xjo, xjoz. These eight inputs were chosen to represent two examples from each possible code execution path with the minimum possible input length. Figure 12 displays the resulting FFT of a high-pass filtered (HPF) set of these captures. Eight clock harmonics can be seen at 4, 8, 16, 32, 64, 128, 256, 512 multiples before petering out at the oscilloscopes internal low pass filter. The 96 MHz fundamental clock frequency is now the strongest component.

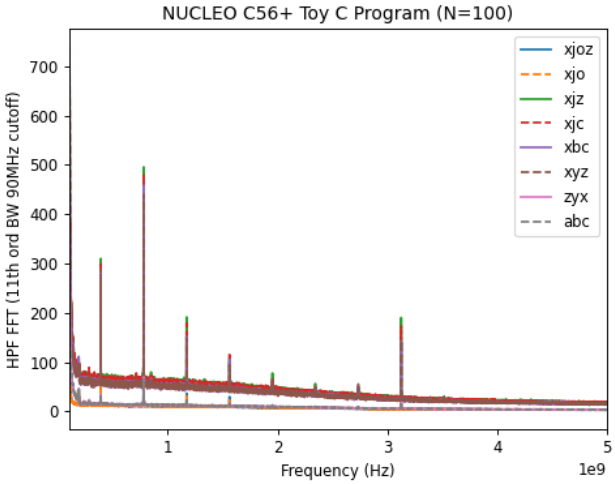

Figure 12. (U) The frequency components of a set of toy program executions representing all four unique code execution paths, only at frequencies above the clock.

(U) Looking at these higher-than-clock frequencies, which are not correlated to program execution duration, we visualize the uniqueness of side-channel captures for different amounts of averaging in Figure 13. It appears that the second conditional branch will be the hardest to distinguish, and we hypothesize averaging would be required for this branch to be accurately classified. The first and third branch should be trivial to distinguish, and should not require averaging. This variation in the difficulty to detect frequency-domain features unique to different basic blocks of real programs prompts further study, which we leave to future work. Our toy C program could be either easier or harder to analyze, depending on the diversity and variability of instructions in the code traces.

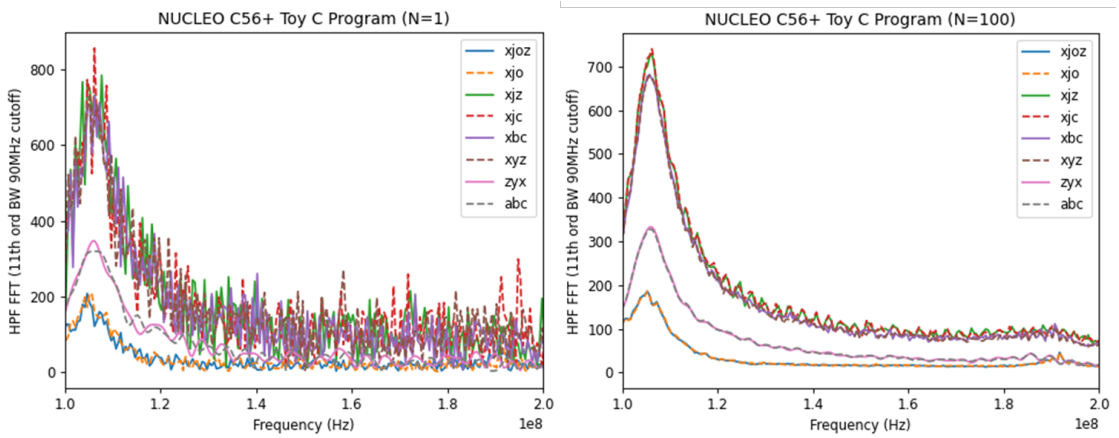

Figure 13. (U) The frequency components of a set of toy program executions representing all four unique code execution paths, for 1 execution each (left) and 100 (right).

(U) We obtain inference results (Figure 14) for first 8, then 75 unique program inputs, requiring 15 bytes of memory per input and collecting at a rate of roughly 8.13 executions/second. It can be seen that the SDIV instruction cluster yields the largest distance variance, which we found to group well into positive and negative divisor clusters. For N=1 executions/input on the NUCLEO, we require 0.033 seconds to analyze that data for coverage, achieving an F1 score of 0.875, where conditional branch two is not detected (i.e., a false negative occurs). On the CW308T-STM32F, we did not require averaging to achieve a score of 1. By increasing N to 15, we require 20 seconds to collect data and again 0.033 seconds to analyze, achieving an F1 score of 1 on the NUCLEO. This confirms our hypothesis that a small amount of averaging is required to determine if conditional branch two was taken or not, and shows that side-channel emissions at higher-than-clock frequencies can correlate to code execution path. We hypothesize the need for averaging is driven by the architecture complexities outlined in Section 3, since this much averaging was not required for the simpler CW architectures running the same toy C program. Our choice of program inputs demonstrates that our algorithm does not over-fit, since we execute on multiple unique inputs per CFG path. From OCSVM, ECOD, and KS test anomaly detectors, we found KS tests to work the fastest and most accurately, with a significance level of 1% ($\alpha = 0.01$).

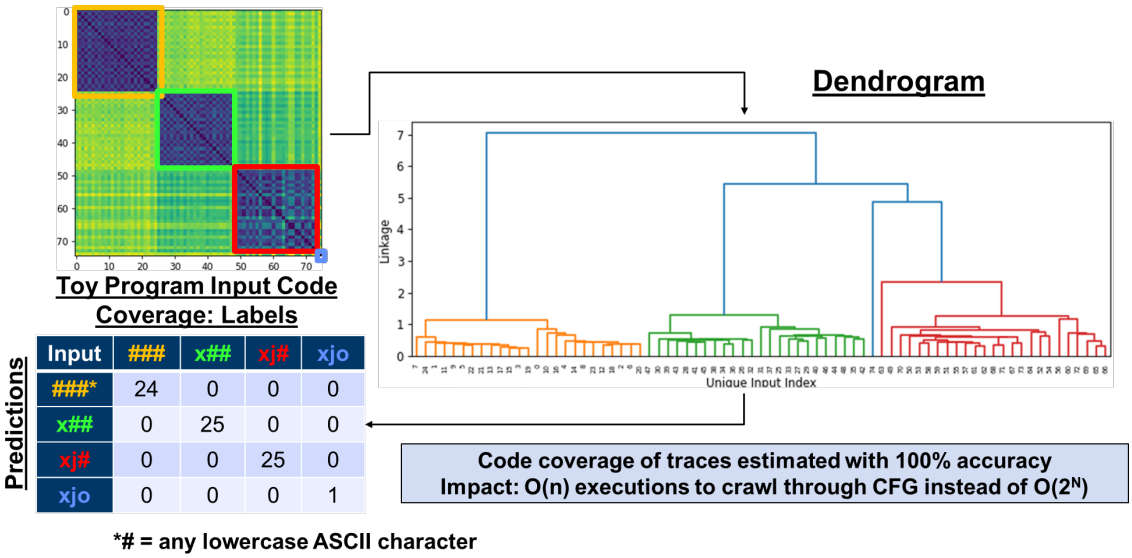

Figure 14. (U) Side-channel clock cycle inference data's pairwise distance for 75 unique toy C program inputs executed by the ChipWhisperer CW308T-STM32F is displayed as a matrix and a dendrogram. Distance is computed using relative cluster labels, such that if the same cluster in predicted and true sets has different integer IDs, it will not increase distance score. Thresholding by the longest distance split, side-channels are correctly grouped by CFG path. Experiment is repeated with the NUCLEO target instead of the CW, with equal classifications seen in the confusion matrix.

(U) We then fuzzed our toy program on the NUCLEO to demonstrate to ourselves what the SCARFES system could look like, and stress these results are not representative. We leave the fuzzing of a corpus of benchmark programs as future work. In this experiment, the SCARFES

software received inputs from AFL++ instead of from our eight curated program inputs. We initialize with a text file seed containing "y". When AFL++ is run, a live grid of statistics is displayed on the console window (Fig. 15). By watching the grid for a few minutes, we can do a number of checks. A "last new find" within the first few minutes of the experiment is a good sign, and we see one after 55 seconds. Our instantaneous execution speed of roughly 5.41 executions per second is a fairly standard on-chip fuzzing rate, indicating our inference is not slowing down the experiment too much. Finally, the low (less than 2.00%) map density of 0.00%/0.00% communicates that the program is very simple, which it is. The stability indicates how often our inference returns the same value, and for our high-stability methodology we see a value of 100%. We see the corpus has a size of 3 because the inputs "x\n" and "xJ" have been added to the seed of "y", since they resulted in instrumented coverage with novel edges.

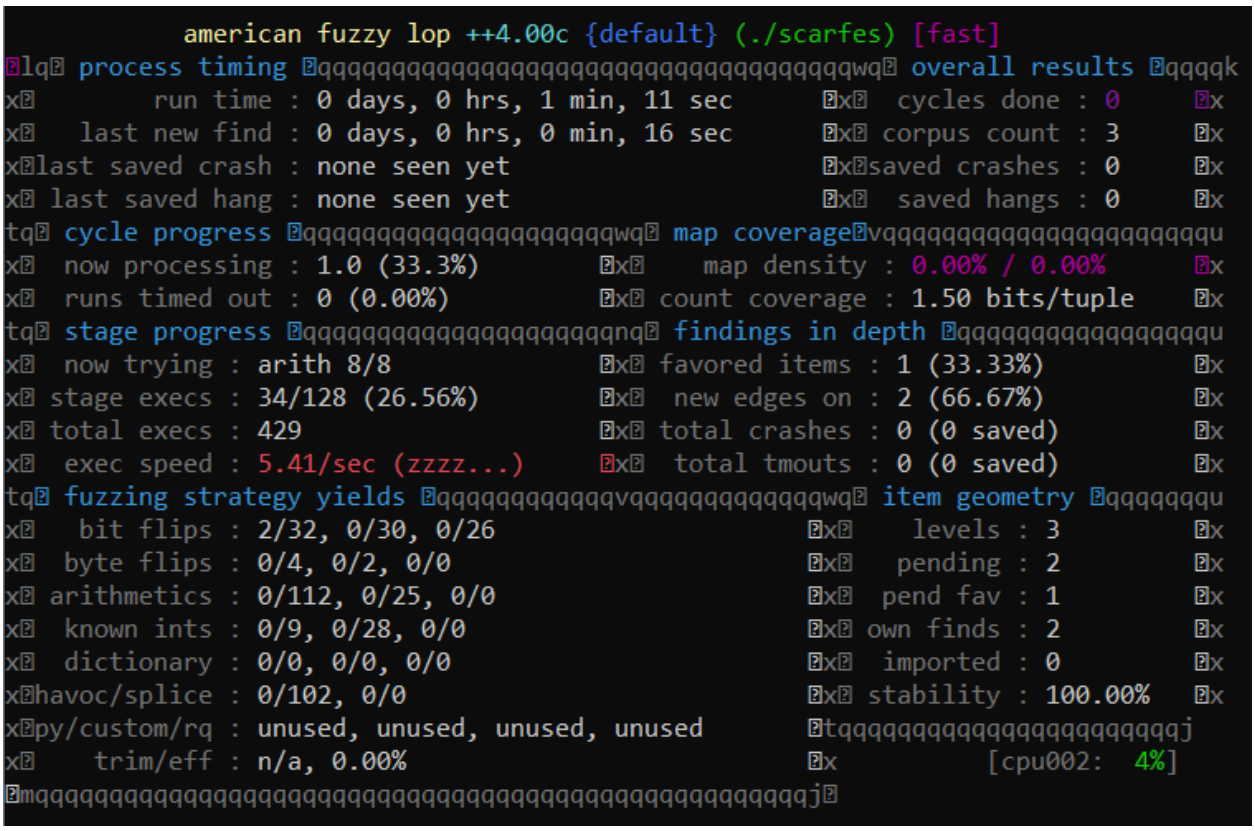

Figure 15. (U) A screen shot of the AFL++ status screen during on-chip fuzzing in instrumentation mode

(U) Coverage curves (Fig. 16) are a classic deliverable for fuzzing projects, where the horizontal axis is time and the vertical axis is some coverage metric such as map density ratio. We

present a coverage curve result to demonstrate scaled coverage inference, where results thus far show small-scale results on curated program inputs. Coverage curves are typically averaged over multiple experiments since AFL++ does most of its mutations in a random "havoc" mode. Unfortunately, we did not have time to average the top two tiers of coverage discovery due to the slow, serialized nature of on-chip fuzzing, so our result is purely for demonstration purposes and may not repeat the same way. We provide three experimental modes: instrumented, which acts as an upper bound on performance, side-channel mode, which demonstrates our novel methodology, and random mode, which serves as a lower bound on performance. Fuzzing for all modes is performed with deterministic mode enabled and a timeout of 10,000 executions/cycle. These results demonstrate an example of how SCARFES might benefit VA with faster-than-random coverage discovery without the use of laborious instrumentation efforts. The first tier of coverage discovery, "branch 1 not taken" takes as long to discover as a single execution. The second tier is extremely consistent across all three modes since it is discovered during deterministic mutations. The third and fourth tier of coverage discovery are where havoc mutations are discovering new coverage, and it is here where we begin to see gains. Our side-channel methodology matched instrumented speeds for discovering this branch, roughly 60x faster than random. The fourth tier of coverage was discovered roughly 8x slower than instrumented. Random mode did not discover this branch in the time we had, so the value listed is when we ended the session. Given simulations using hard-coded Ghidra basic block data, we estimate this value would've been about 84 hours at a more optimized random speed of 100 executions/second, which would've been roughly 35x slower than SCARFES despite executing roughly 15x more times per second.

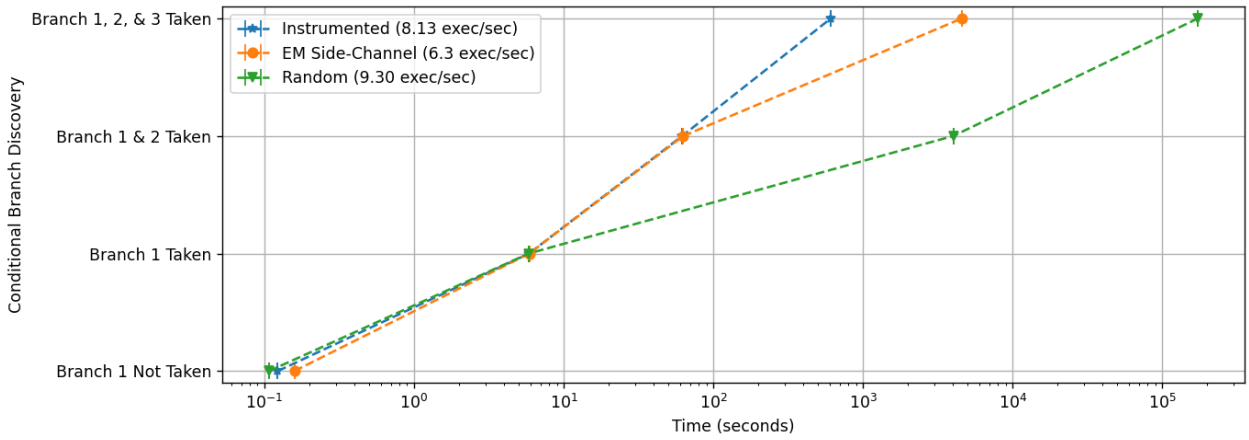

Figure 16. (U) A simple experiment that visualizes how SCARFES could benefit ES VA by fuzzing for coverage faster than random but without the labor of developing instrumentation.

This page intentionally left blank.

5. (U) DISCUSSION AND FUTURE WORK

(U) In this report, we demonstrated the feasibility of CFG inference via EM side-channel measurements for the purpose of increasing the speed of coverage discovery in ES on-chip fuzzing for a more realistic embedded processor than has been previously studied in SBCD research. Most importantly, our methodology does so without the use of fingerprinting methodologies, instead utilizing unsupervised approaches. Section 4 demonstrates an example of how inferred coverage can be used to achieve faster-than-random fuzzing CFG exploration without the need for rehosting.

(U) We highlight here future work for the reader to consider for research on the topic of unsupervised EM side-channel fuzzing.

- Program execution start and stop detection via side-channel data. We hypothesize this could be done via high SNR side-channel signals such as on-chip peripheral communications (e.g., cryptography rounds [12]), off-chip peripheral (e.g., UART) communications, and "idling" fingerprinting (i.e., what the loop side-channel looks like when no input is received).
- Over time, one could identify a common conditional branch pattern of instruction IDs (e.g., block $4 \rightarrow 5 \rightarrow 4 \rightarrow 5$ in Figure 3), and therefore detect branch points before both paths have been exercised.
- Some devices perform speculative execution, meaning they will pre-load the pipeline with instructions that may or may not be correctly predicted, necessitating pipeline flushes in the case of incorrect speculations. One might leverage this to identify branch instructions.
- Program execution crash and hang detection via side-channel data, communicated as a status flag to AFL++ to aid in log organization. We hypothesize hangs could be detected as a subset of the program execution stop problem (i.e., detect when the program hasn't stopped by some timeout duration), and that crashes could be detected by fingerprinting the reboot EM side-channel sequence, since bare-metal ES architectures reboot when a crash such as a FPE occurs.
- Fuzzing programs with a greater number of branches and which utilize a larger percentage of the ARM instruction set.
- Fuzzing more complex ES architectures such as those with operating systems, multiple MCUs, peripherals, etc. We hypothesize that added architectural complexity will require more averaging, processing, and inference algorithms, which reduce the feasibility of such targets for side-channel fuzzing. We hypothesize this methodology will be most useful for simpler architectures where instruction execution is strongly correlated to EM side-channel leakage.

This page intentionally left blank.

(U) GLOSSARY

AFL++	American Fuzzy Lop
ALU	Algorithmic Logic Unit
ARM	Advanced RISC Machine
CDF	Cumulative Distribution Function
CFG	Control Flow Graph
DoD	Department of Defense
DPA	Differential Power Analysis
ECOD	Unsupervised Outlier Detection Using Empirical Cumulative Distribution Functions
ES	Embedded Systems
FPE	Floating Point Exception
GPIO	General-Purpose Input/Output
IC	Integrated Circuit
IDE	Integrated Development Environment
IF	Isolation Forest
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
KS	Kolmogorov-Smirnov
MCU	Microcontroller Unit
OCSVM	One-Class Support Vector Machine
PCB	Printed Circuit Board
RE	Reverse Engineering
RF	Radio Frequency
RISC	Reduced Instruction Set Computer
SCARFES	Side-Channel Assisted Real-time Fuzzing for Embedded Systems
SMD	Surface-Mounted Device
UART	Universal Asynchronous Receiver-Transmitter
UNIX	UNiplexed Information Computing System
UTF	UCS (Unicode) Transformation Format
VA	Vulnerability Assessment

(U) GLOSSARY
(Continued)

VDD	Voltage Drain

(U) REFERENCES

[1] (U) A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson, "Sok: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, New York, NY, USA: Association for Computing Machinery (2021), ASIA CCS '21, p. 687701, URL https://doi.org/10.1145/3433210.3453093, UNCLASSIFIED.

[2] (U) J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Computing Surveys* 55(7), 1–33 (2022), UNCLASSIFIED.

[3] (U) M. Chilenski, C. Cleveland, I. Dekine, C. O'Donnell, G. Raz, A. Sciotti, and L. Vertatschitsch, "Identifying class of previously unseen programs using rf side channels," in *Cyber Sensing 2020*, SPIE (2020), vol. 11417, pp. 36–46, UNCLASSIFIED.

[4] (U) P. Sperl and K. Böttinger, "Side-channel aware fuzzing," in *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, Springer (2019), pp. 259–278, UNCLASSIFIED.

[5] (U) S. Nilizadeh, Y. Noller, and C.S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE (2019), pp. 176–187, UNCLASSIFIED.

[6] (U) Y. Noller and S. Tizpaz-Niari, "Qfuzz: Quantitative fuzzing for side channels," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021), pp. 257–269, UNCLASSIFIED.

[7] (U) K. Ramezanpour, P. Ampadu, and W. Diehl, "Scaul: Power side-channel analysis with unsupervised learning," *IEEE Transactions on Computers* 69(11), 1626–1638 (2020), UNCLASSIFIED.

[8] (U) M.T. Rahman, D. Forte, J. Fahrny, and M. Tehranipoor, "Aro-puf: An aging-resistant ring oscillator puf design," in *2014 design, automation & test in Europe conference & exhibition (DATE)*, IEEE (2014), pp. 1–6, UNCLASSIFIED.

[9] (U) A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association (2020), UNCLASSIFIED.

[10] (U) J.C. Maxwell, "On physical lines of force," *Philosophical magazine* 90(S1), 11–23 (2010), UNCLASSIFIED.

[11] (U) G.H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal* 34(5), 1045–1079 (1955), UNCLASSIFIED.

[12] (U) S.J. Wang, Y.C. Lin, C.Y. Lin, S.K. Chong, and K.S.M. Li, "Automatic alignment for side-channel attack against cryptographic modules," in *2022 IET International Conference on Engineering Technologies and Applications (IET-ICETA)* (2022), pp. 1–2, UNCLASSIFIED.

[13] (U) V. Cristiani, M. Lecomte, and T. Hiscock, "A bit-level approach to side channel based disassembling," in *Smart Card Research and Advanced Applications: 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11–13, 2019, Revised Selected Papers 18*, Springer (2020), pp. 143–158, UNCLASSIFIED.

[14] (U) S. Vafa, M. Masoumi, and A. Amini, "An efficient profiling attack to real codes of pic16f690 and arm cortex-m3," *IEEE Access* 8, 222520–222532 (2020), UNCLASSIFIED.

[15] (U) J. Park and A. Tyagi, "Using power clues to hack iot devices: The power side channel provides for instruction-level disassembly." *IEEE Consumer Electronics Magazine* 6(3), 92–102 (2017), UNCLASSIFIED.

[16] (U) J.A. Hartigan and M.A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)* 28(1), 100–108 (1979), UNCLASSIFIED.

[17] (U) S. Alnegheimish, D. Liu, C. Sala, L. Berti-Equille, and K. Veeramachaneni, "Sintel: A machine learning framework to extract insights from signals," in *Proceedings of the 2022 International Conference on Management of Data*, Association for Computing Machinery (2022), SIGMOD '22, p. 18551865, UNCLASSIFIED.

[18] (U) L. Ruff, R. Vandermeulen, N. Goernitz, L. Deecke, S.A. Siddiqui, A. Binder, E. Müller, and M. Kloft, "Deep one-class classification," in J. Dy and A. Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, PMLR (2018), *Proceedings of Machine Learning Research*, vol. 80, pp. 4393–4402, URL https://proceedings.mlr.press/v80/ruff18a.html, UNCLASSIFIED.

[19] (U) F.T. Liu, K.M. Ting, and Z.H. Zhou, "Isolation forest," in *2008 eighth ieee international conference on data mining*, IEEE (2008), pp. 413–422, UNCLASSIFIED.

[20] (U) L.M. Manevitz and M. Yousef, "One-class svms for document classification," *Journal of machine Learning research* 2(Dec), 139–154 (2001), UNCLASSIFIED.

[21] (U) F.J. Massey, "The Kolmogorov-Smirnov test for goodness of fit," *Journal of the American Statistical Association* 46(253), 68–78 (1951), UNCLASSIFIED.

[22] (U) Z. Li, Y. Zhao, X. Hu, N. Botta, C. Ionescu, and G.H. Chen, "ECOD: unsupervised outlier detection using empirical cumulative distribution functions," *CoRR* abs/2201.00382 (2022), URL https://arxiv.org/abs/2201.00382, UNCLASSIFIED.

[23] (U) J. van Woudenberg and C. O'Flynn, *The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks*, No Starch Press (2021), URL https://books.google.com/books?id=DEqatAEACAAJ, UNCLASSIFIED.