Glitching Demystified: Analyzing Control-flow-based Glitching Attacks and Defenses

Chad Spensky*¶‖, Aravind Machiry†, Nathan Burow‡, Hamed Okhravi‡, Rick Housley§, Zhongshu Gu¶, Hani Jamjoom¶, Christopher Kruegel‖, Giovanni Vigna‖
*Allthenticate †Purdue University ‡MIT Lincoln Laboratory §River Loop Security
¶IBM T.J. Watson Research Center ‖UC Santa Barbara
chad@allthenticate.net, amachiry@purdue.edu, {nathan.burow, hamed.okhravi}@ll.mit.edu,
rick@riverloopsecurity.com, {zgu, jamjoom}@us.ibm.com, {chris, vigna}@cs.ucsb.edu

***Abstract*—Hardware fault injection, or *glitching*, attacks can compromise the security of devices even when no software vulnerabilities exist. Attempts to analyze the hardware effects of glitching are subject to the Heisenberg effect and there is typically a disconnect between what people "think" is possible and what is actually possible with respect to these attacks. In this work, we attempt to provide some clarity to the impacts of attacks and defenses for control-flow modification through glitching. First, we introduce a glitching emulation framework, which provides a scalable playground to test the effects of bit flips on specific instruction set architectures (ISAs) (*i.e.,* the fault tolerance of the instruction encoding). Next, we examine *real* glitching experiments using the ChipWhisperer, a popular microcontroller using open-source glitching hardware. These real-world experiments provide novel insights into how glitching attacks are realized and might be defended against in practice. Finally, we present GLITCHRESISTOR, an open-source, software-based glitching defense tool that can automatically insert glitching defenses into any existing source code, in an architecture-independent way. We evaluated GLITCHRESISTOR, which integrates numerous software-only defenses against powerful and real-world glitching attacks. Our findings indicate that software-only defenses can be implemented with acceptable run-time and size overheads, while completely mitigating some single-glitch attacks, minimizing the likelihood of a successful multi-glitch attack (*i.e.,* a success rate of 0.000306%), and detecting failed glitching attempts at a high rate (between 79.2% and 100%).**

I. INTRODUCTION

Hardware-induced faults [32], which we refer to as *glitches*, are capable of corrupting the system state by modifying both instructions and data, and can be leveraged to undermine software-based security mechanisms, even if the software security mechanisms are implemented *with no semantic vulnerabilities*. Indeed, malicious glitches have been leveraged to compromise secure smartcards [12], [7], [6], security-hardened gaming consoles (*e.g.,* the XBOX 360 [59], Playstation 3 [38], Playstation Vita [44], and Nintendo Switch [68], [26]), and enterprise Internet protocol (IP) phones [19]. Glitching attacks have even been leveraged to bypass both Intel's Software Guard Extension (SGX) protections [51] and ARM's TrustZone [67] and even extract hardware-embedded cryptographic keys [42]. However, little has been done to adequately study and defend against these types of attacks in practice. Some code-level glitching mitigations [82] have been proposed, but have not had their underlying assumptions or efficacy evaluated on real-world systems. Alternatively, custom-built hardware-based counter-measures (*e.g.,* brownout detection or lock-step computation) [20] are currently only sparsely deployed, due to cost and complexity, leaving the majority of embedded systems susceptible to glitching attacks.

Glitching attacks involve introducing a physical disturbance to a system that will ultimately corrupt the instructions being executed or the data being manipulated. This corruption can be achieved by changing the supply voltage [43], [13], optical probing with lasers [71], [79], disrupting the clock [4], or introducing an electromagnetic pulse (EMP) [57], [48]. To leverage these faults in a successful attack, the fault must be injected at a specific time in the execution pipeline. For example, if the execution was corrupted precisely when a security-critical branch condition was being checked (*e.g.,* checking the kernel's signature [22]), that instruction could be changed to a *no operation* instruction, and effectively skipped, allowing the attacker to disable secure boot [19], [77], escalate privileges [76], or extract "protected" code [44].

While effective defenses against other physical attacks are becoming commonplace in commodity computing systems (*e.g.,* trusted boot and encrypted memory), glitching defenses are still lacking. We hypothesize that this is likely due to a general lack of understanding about what exactly glitching attacks are capable of, and, subsequently, a systematic way to implement defenses against them. Indeed, we have observed a large disconnect between theory and practice in this field. For example, many researchers believe that glitching is capable of changing any pointer (*e.g.,* the program counter) in memory or making arbitrary code modifications because of published papers demonstrating this [76], [78], [30]. However, these effects are only realistic in laboratory environments with systems that are well understood and have already had the appropriate glitching parameters "tuned." For all intents and

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. 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.

purposes, these types of attacks are impossible in practice.

In this work, we introduce an open-source QEMU-based glitching emulation environment. This framework was used to exhaustively evaluate an ISA's instruction encoding against specific glitching effects (*e.g.,* bit flips), and examine the result of those instruction-level effects against a program's control flow. These flipped bits ultimately change the instruction being executed or the data being evaluated in a way that is beneficial to the attacker. In fact, our analysis confirmed that by simply flipping bits, the glitch can effectively "skip" an instruction with a high likelihood (*i.e.,* changing the targeted instruction into a *no operation*). We also found that this effect is often non-uniform. For example, on 16-bit ARM processors, glitches that tend to flip bits from 1 to zero appear to be exceptionally powerful (*i.e.,* "skipping" all branch instructions more than 60% of the time), while glitches that flip zeros to ones were less so (*i.e.,* "skipping" branches less than 30% of the time).

In addition to emulating glitches, we also used a popular glitching tool (*i.e.,* the ChipWhisperer [54]) to conduct a suite of real-world glitching experiments to examine the effects of glitching on control-flow-related instructions and data. In particular, our experiments were focused on using glitching to evade guard conditions. This evasion could be used to bypass security-critical code (*e.g.,* verifying signed code, disabling a debug interface, or checking user permissions). Our real-world glitching results provide new insights into how this corruption ultimately affects control flow. For example, load and store instructions appear to be more susceptible to glitching; the value being compared affects the *glitchability* of a branch condition (*e.g.,* `while(!a)` is more vulnerable than `while(a)`); and instructions which simply manipulate registers (*e.g.,* addition) appear to be exceptionally difficult to glitch. We leverage these findings to build our defense framework.

We present the first automated, open-source glitching defense framework, GLITCHRESISTOR, which is capable of adding various glitching defenses at compile time to *any* source code in an architecture-independent way. GLITCHRESISTOR implements numerous proposed glitching defenses (*e.g.,* double checking branches and loop guards, injecting random timing, and integrity checking on sensitive variables). We used GLITCHRESISTOR, combined with our ChipWhisperer-based glitching framework to evaluate the efficacy of these defenses in practice, examining their ability to thwart glitching, as well as the size and run-time overheads that each incurs. GLITCHRESISTOR was able to successfully defend against, and detect, every single-glitch attack that we attempted in our evaluation, necessitating a successful *multi-glitch* attack (*i.e.,* a glitch that affects multiple clock cycles) to evade the implemented defenses. Even so, GLITCHRESISTOR was able to reduce the success rate of our most powerful, multi-glitch attack to 0.263% in the worst case and 0.00306% in the best case, with detection rates of 79.2% and 99.7% respectively.

In summary, we make the following contributions:

- a comprehensive analysis of glitching attacks and their effects on control flow,
- a framework for emulating glitching attacks,
- a breadth of glitching experiments that characterize the effects of glitching and demonstrate the effectiveness of various software-only defenses,
- GLITCHRESISTOR (https://github.com/ucsb-seclab/glitch_resistor), the first extensible glitching defense tool for automatically protecting vulnerable code, and
- an evaluation of GLITCHRESISTOR on real hardware, which demonstrates the effectiveness of software-only defenses, minimizing the likelihood of a successful attack and effectively detecting all glitching attempts in practice.

II. BACKGROUND

Fault injection is well-studied in the context of ensuring the reliability of a computer system [32]. Both software [24] and hardware [5] induced faults are capable of modifying the state of a system and disrupting its typical execution. Indeed, the act of inducing malicious software faults, which materialize as software bugs and vulnerabilities, has spawned an entire subfield of bug finding [72] and fuzzing techniques [1]. In contrast, malicious hardware-induced hardware faults were widely ignored by the software community until the relatively recent exposure of Spectre [34], Meltdown [41] (microarchitecture attacks), and Rowhammer [70], [33], an attack against dynamic random access memory (DRAM). Malicious physical hardware-induced faults are still relatively unexplored.

Hardware-based attacks can be done either invasively (*e.g.,* decapsulating the chip [29]) or non-invasively (*e.g.,* through electromagnetic interference [56]). Non-invasive glitching techniques allow an attack to go undetected and typically permit the attacker to repeat the attack indefinitely. The general idea behind glitching is to interfere with a Flip-Flop circuit, transistor, or capacitor's normal operation to change the stored value or the execution's output. This can be done using any form of interference, be it an external physical phenomenon, like temperature or electromagnetic (EM) interference, or by operating the system outside its designed conditions (*e.g.,* by modifying the voltage or clock). In practice, voltage glitching, which is done by either increasing or decreasing the voltage for a brief period of time, and clock glitching, which involves inserting additional clock edges, are the most common glitching techniques, due to their relatively low cost and their effectiveness.

In this work, we only examine *non-invasive attacks*, as defenses against invasive attacks [23] necessarily require hardware modifications. For those interested in the specific effects of each type of non-invasive glitch, we refer the reader to Section 5.3 of the resulting dissertation [73].

A. Motivation

Glitching attacks have already been used to attack numerous commercial systems. For example, researchers were able to use glitching to defeat the security on two automotive safety integrity level (ASIL)-D[1] compliant automotive microcontroller units (MCUs) [81], evading hardware-based countermeasures

[1]The most stringent ASIL requirements of safety and fault tolerance.

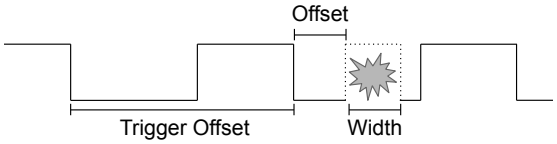


Fig. 1: The three parameters that need to be tuned for clock glitching: the offset from the trigger, the offset into the clock cycle, and the width of the injected clock cycle

like Flash error-correcting code (ECC) and lockstep execution, using EM and voltage glitching, respectively. The same researchers were also able to bypass authentication checks, and even re-enable the Joint Test Action Group (JTAG) interface. Similarly, voltage glitching has also been used to extract both Rivest, Shamir, and Adleman (RSA) [8], [69] and advanced encryption standard (AES) [9], [42] keys, and has even been shown to be effective against programs executing on modern Android phones and the Raspberry Pi, both running Linux [53]. More powerful attacks have even been able to control the program counter (PC) directly with glitching [78], [30]. In the case of defeating a secure boot loader, which has a relatively small attack surface and takes little or no user input, glitching attacks are one of the only methods for compromising the boot loader's security.

B. "Tuning" the Glitch

All glitching techniques necessarily require a "tuning" phase where the location and specific glitching parameters are tweaked until the desired effect is achieved. The attacker must first figure out *when* to inject the glitch, by calculating an offset from a known *trigger* (*i.e.,* an observable artifact that indicates which code is currently executing). For example, to inject a clock glitch, an attacker must simultaneously configure both the width and location in the clock cycle to inject a glitch, as well as the offset from an observable trigger (see Figure 1). Similar parameters must be tuned for both voltage and EM glitches (*e.g.,* the duration and voltage of the attack or the location and intensity of the EMP).

In our ideal laboratory environment with a perfect trigger, we were able to consistently, and automatically, tune our clock glitching parameters and successfully glitch an unprotected embedded system 100% of the time (10 out of 10 attempts) in less than 16 minutes, in the best case. However, this is only possible with an initial search over the parameter space, which is the exact step that our evaluated defenses are targeting.

C. Defenses

Hardware-based defenses typically involve inserting additional circuits (*e.g.,* to detect voltage glitches [83]), an additional run-time monitor [65], [3], or control-flow integrity (CFI) signatures [66], [80]. However, hardware modifications are impractical for the many already-deployed Internet of things (IoT) devices. They are also far less likely to be adopted for individual systems, due to the lead times on hardware fabrication. Therefore, software-based techniques are more likely to be useful as practical defenses.

Software-based glitching defenses can never completely mitigate the problem. In the limit, glitching could (in theory) be used to skip *every* defensive instruction and even transform benign instructions into malicious ones. Nevertheless, software-based techniques are cheaper to implement and can be effective at defending against real-world attacks (in practice) by making the required scenario for a successful glitching attack increasingly improbable. Unfortunately, existing techniques, which rely on redundancy [49], only work on simple code-bases and have simplistic attacker models, which makes them infeasible on real-world code.

III. Threat Model

Non-invasive glitching attacks require physical access to the device being glitched and control over the specific input being glitched (*e.g.,* the voltage line, clock line, or access to microchip). An attacker can dismantle any external packaging (*e.g.,* remove the case containing the electronic components), but cannot modify the electronic components in any non-reversible way. For example, an attacker may solder a wire to a specific pin to bypass a voltage regulator, but cannot remove or modify the integrated circuit (IC) directly.

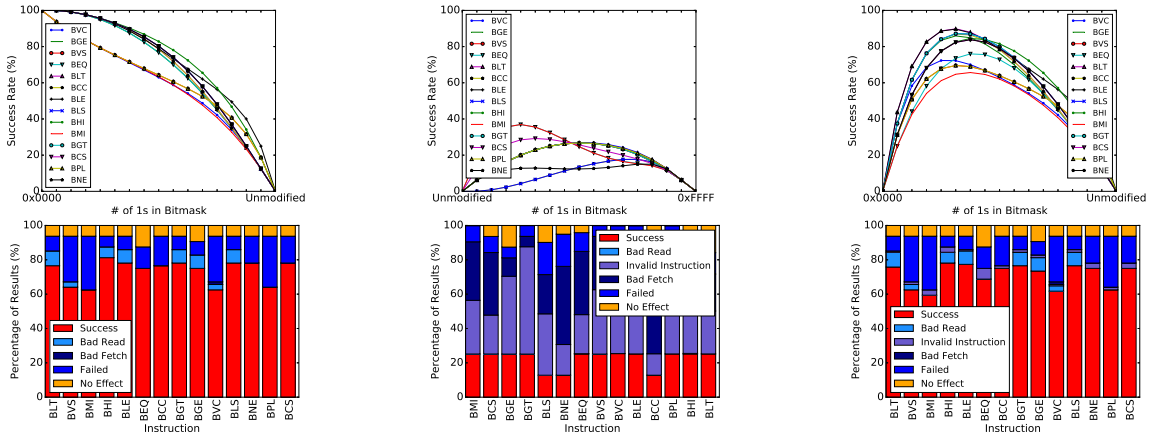
This threat model is realistic for any deployed embedded system: IoT devices, gaming systems, automobiles, robots, or military drones. The goal is typically to either bypass integrity checking of the firmware or extract the firmware image for reverse engineering. As previously mentioned, the system must necessarily have some externally observable trigger to create a reliable glitch (*e.g.,* a voltage dip, an observable output, or a request for user input). In the various high-profile glitching attacks against gaming systems [59], [38], [44], [68], [26], the exploits were crafted by first identifying the *approximate* area that appeared to be vulnerable (*e.g.,* right before an error code) and then tuning the glitching parameters (*e.g.,* clock waveform, voltage modification, or EM power and position). No two systems are physically identical, which means that each attack must be specialized for the specific system being attacked. Even commercialized attacks (*e.g.,* the XBOX reset attack) are typically probabilistic, due to physical limitations, and have some method for automatically retrying the glitch in the event of a failure.

IV. Glitching Effects in Emulation

To gain a better understanding of the theoretical limit on the effectiveness of glitching, we first investigate the following research question:

RQ1 What is the likelihood that random bit flips will result in a "skipped" control-flow instruction?

To quantify the effects of bit flips on a specific ISAs, we built an emulation framework that is capable of forcing bit flips (*i.e.,* corrupting specific instructions) and executing the resulting code to determine the effects on the control flow of the program. Previous literature [35], [64], [48], [78], [74], [4] indicates that bit flips induced by glitching tend to be unidirectional (*i.e.,* either flipping 1s to 0s or 0s to 1s, but not both). While complex bit flips are possible, they are



(a) AND (b) OR (c) AND (`0x0000` Invalid)

Fig. 2: The probability of a glitch succeeding on ARM Thumb as a function of the number of bits that were flipped and how they were flipped, *i.e.,* 1s to 0s (`AND`) or 0s to 1s (`OR`), computed by taking every possible combination, *i.e.,* $\binom{n}{k}$, of bits for each flip value and creating a bit mask that was either ANDed or ORed with the original instruction. The reasons for the failures are shown in the accompanying histograms.

improbable in practice [78]. Therefore, we only present the results for unidirectional flips for our evaluation (*i.e.,* logical `and` and `or` operations). We also tested bidirectional bit flips (*i.e.,* `xor`), and the results were in between the those of `and` and `or`, which are shown here.

We implemented our glitch emulator using Unicorn [52] for central processing unit (CPU) emulation, Capstone [61] for disassembling code, and Keystone [62] for assembly. All of our test cases are manually written for the instruction in question such that a successful glitch (*i.e.,* the targeted instruction was skipped) will place the value `0xdead` in a known register, and a normal execution will place the value `0xaaaa` in a separate known register. Because these snippets of code are so small (*e.g.,* 3-5 lines of assembly), we are able to completely isolate the instruction in question. Our automated framework takes this source code, assembles it to machine code, and then generates *every* possible bit mask for *every* possible number of bits. More precisely, it produces $\binom{n}{k}$ possible bit masks for each k, where n is the number of bits in the instruction and k is the number of bits being mutated. These bit masks are then either `AND`ed or `OR`ed with the target instruction and then the entire program is executed in an emulator. Upon completion, the register values and error codes are read to log the result.

We used this framework to quantify the effects of glitching on the popular 16 bit ARM Thumb architecture. The results for *every* conditional branch instruction in ARM Thumb under the `AND` and `OR` perturbation conditions can be seen in Figure 2. In these figures, a glitch is considered a "success" if the instruction immediately following the conditional branch, which would otherwise not be executed, was executed successfully. The failures are grouped in the following way: a *bad read* is when the system attempted to read unmapped memory; an *invalid instruction* is thrown when the emulator did not recognize the perturbed instruction; a *bad fetch* is thrown when an instruction was fetched from unmapped memory (*e.g.,* the PC was modified); an unknown *failure* is any unrecognized error; and, if the modification had *no effect* on the execution of the code, we annotate it as such.

One immediate observation is that the `AND` model exhibits a substantially higher success rate than the `OR` model. Initially we hypothesized that this was because in our experiments, the conditional branches had a relatively low Hamming weight (*e.g.,* `beq #6` is repressed as `0b1101 0000 00000000` [2]), and thus converting them all to zeros, which is interpreted as `mov r0, r0` (or *no operation*) in ARM Thumb, was highly likely. However, after modifying our emulator to interpret all 0s as an *invalid instruction* this hypothesis was quickly debunked since the overall success rate was effectively unchanged (see Figure 2c). Thus, it appears that the ISA itself is simply vulnerable to glitches that are capable of flipping 1s to 0s, which is, unfortunately, the most likely effect of the cheaper, more popular forms of glitching (*i.e.,* voltage and clock). Nevertheless, in practice, we hypothesize that a minor modification like this to the ISA could pay large dividends. Similarly, adding invalid instructions in between valid instructions would likely thwart many glitching attempts. However, the only way to test these hypotheses against physical glitches would be to fabricate a microchip with a modified ISA, which is out of scope for this work.

V. Real-world Glitching

To glean insights into real-world glitching effects, we employed the popular open-source ChipWhisperer Lite, a suite of hardware and software tools that enable glitching and side-channel analysis. In our experiments, we wanted to evaluate the *upper bound* of glitching effectiveness (*i.e.,* the best case scenario for an attacker, and the worst case scenario for the system being glitched). Therefore, we used

TABLE I: The number of successful glitches for each clock cycle, mapped to the respective instruction that was executing and with a post-mortem view of the comparator register

Cycle	Instruction	Successes	R3	Count
0	`MOV R3, SP`	110	0	44
			8	32
			0x21	33
			0x68	1
1	`ADDS R3, #7`	9	8	8
			0xFF	1
2	`LDRB R3, [R3]`	-	-	-
3		18	0	18
4	`CMP R3, #0`	43	0	1
			8	37
			0x55	2
			0x20003FE8	3
5	`BEQ .loop`	89	8	41
			0x55	4
			0x20003FE8	44
6		133	8	49
			0x55	3
			0x20003FE8	73
			0x20003FEF	2
			0x28004309	6
7		183	0	41
			8	102
			0x20003FE8	36
			0x28004309	1
			0x40007FD7	1
			0xDFFFC010	1
			0xFFFFFFF9	1
	Total	585 (0.705%)	12 unique	

(a) `while(!a)`, R3=0x1000

Cycle	Instruction	Successes	R3	Count
0	`MOV R3, SP`	84	0	11
			1	38
			0x55	33
			0x68	1
			0xFF	1
1	`ADDS R3, #7`	14	0	4
			0x55	10
2	`LDRB R3, [R3]`	-	-	-
3		-	-	-
4	`CMP R3, #0`	-	-	-
5	`BNE .loop`	9	0	9
6		39	0x55	32
			0x20003FF6	1
7		126	0	4
			1	39
			8	1
			0x55	82
	Total	272 (0.347%)	7 unique	

(b) `while(a)`, R3=0x1000

Cycle	Instruction	Successes	R2	Count
0	`LDR R2,[SP,#0x10+a]`	25	0	1
			0x4EE6BB18	1
			0xE7D25763	23
1		-	-	-
2		-	-	-
3	`LDR R3,=0xD3B9AEC6`	1	0xE7D25763	1
4	`CMP R2, R3`	1	0xD3B9AEC6	1
5	`BNE .loop`	46	0xD3B9AEC6	1
			0xE7D25763	45
6		150	0x40	2
			0x400	2
			0xE7D25722	1
			0xE7D25763	145
7		129	0x40	1
			0x400	1
			0xE7D25763	127
	Total	352 (0.449%)	7 unique	

(c) `while(a!=0xD3B9AEC6)`, R2=0x48000028, R3=0x1000

the STM32F071RBT6, a 48 MHz ARM Cortex M0 chip with a 3-stage pipeline, as our target board, and drove the clock directly from the ChipWhisperer (*i.e.,* the most powerful glitching attack proposed by previous work). Similarly, we created a *perfect trigger* for each instruction sequence that we wanted to glitch. More precisely, our trigger would apply voltage to a general purpose input/output (GPIO) pin exactly 1 clock cycle before the targeted instruction, which permitted precise, reliable glitches to be injected. These conditions are *ideal* for an attacker and should provide a reasonable upper bound on the capabilities of glitching attacks. Choosing a more advanced chip, a more complex "real-world" firmware, or different glitching mechanism would likely inflate the findings of GLITCHRESISTOR, since producing successful glitches would be more difficult, especially as code complexity and chip complexity increase. Thus, these smaller, controllable experiments against the most powerful glitching technique are, counter-intuitively, a more stringent analysis of GLITCHRESISTOR than a more modern chip with real firmware. We investigate the following research questions:

RQ2 What is the upper bound of glitching effectiveness?
RQ3 Does the value being compared affect its *glitchability*?
RQ4 How are branches being "skipped" (*i.e.,* which instruction is being corrupted, and in which way)?
RQ5 How much more difficult is a multi-glitch (*i.e.,* a glitch that affects multiple instructions)?

A. Glitching Effects

In theory, the actual value being compared should affect the ability to glitch a certain branch. For example, glitching a 1 into a 0 *should* be easier than glitching $0b1010$ into $0b0101$. To test this, we constructed three distinct experiments to evaluate the following expressions: `while(a)`, where `a=1`; `while(!a)`, where `a=0`; and `while(a!=0xD3B9AEC6)`, where `a=0xE7D25763`. These are all implemented as empty infinite loops, with `volatile` variables so they are not optimized out by the compiler (a successful glitch would *exit* the loop). The hypothesis is that `while(a)` and `while(!a)`, which are common in C code, should be much easier to glitch than values with a large Hamming distance, as they both only require a single bit flip to change the outcome of the conditional branch.

To evaluate the effects of glitching on these three loops, we scanned all of the possible glitching parameters (*i.e.,* the full range of possible widths and offsets) for each clock cycle in question. When compiled, each experiment takes up to 8 clock cycles (the branch instruction can take between 1 and 3 clock cycles). Thus, we varied our clock-cycle offset between 0 and 7, and for each clock cycle ranged the width and offset of the glitch (*i.e.,* $[-49\%, 49\%] \times [-49\%, 49\%]$), resulting in 9,801 glitching attempts per clock cycle. The results of these three experiments, along with value observed in the comparison register, can be seen in Table I.

Our results only partially corroborate our hypothesis, with `while (!a)` being the most vulnerable (0.705% success rate) and the other two achieving comparable success rates (0.347% and 0.449%, respectively). Surprisingly, the case where `a` was initialized to 1, and the condition was `while(a)` was the most resilient to glitching. However, after examining exactly how the glitches were succeeding, a different story emerged. The assembly code for each case, along with the corresponding clock cycles, is also shown in each table. Since the processor being glitched has a three-stage pipeline, it is difficult to determine which instruction, and which portion of the pipeline was affected by the glitch, but the location of the glitch at least bounds the glitch's effects.

For example, the initial clock cycles (0 through 4), which set the values, appeared to be more susceptible to glitching in the simple comparison cases (*i.e.,* `!a` and `a`) than in the complex comparison case. This is likely attributed to the fact that the underlying assembly instructions changed as a result of the comparison (*i.e.,* during the fetch stage). But the fact that the instructions have fewer glitchable clock cycles is still significant. In fact, the case for `while(!a)` by far had the most data corruptions that resulted in the branch condition being satisfied, as any non-zero value would suffice.

To explain some of the values that were observed in the resulting comparison register, we attached a JTAG debugger to the board and examined the state of the system *before* the loop was entered. For every case, `0x20003FE8` is the value of `SP`, `0x48000028` and is the GPIO address that was written to. Thus, `0x40007FD7` is likely a mix of the GPIO address and some corruption (Table Ia). Similarly, for the `while(a)` case, `0x20003FF6` is likely a mix of `SP` and some corruption (Table Ib). Interesting, in the `while(a!=0xD3B9AEC6)` case, 2 of the glitches resulted in the comparison register, `R2`, being correctly set to the unlikely value of `0xD3B9AEC6`, which is not on the stack, but is only stored as intermediate (Table Ic). This must mean that the `LDR` instruction was corrupted to load the valid into the wrong register. Similarly, the various `0x4` values are likely a residual from the address in the register during a load. We were unable to identify any obvious connections to the other values stored in the registers, and can only assume that they are attributed to random flips.

B. Locating Optimal Parameters

We also investigated the *best case* scenario for glitching an unprotected conditional branch. In this experiment, we sought to identify glitch parameters that would have a 100% success rate. To achieve this, our algorithm starts by scanning our glitching parameters (*i.e.,* target offset, width, and offset) with a 10 cycle clock glitch, which encompasses every instruction in the `while` loop. Once successful parameters are identified, the algorithm then tests each individual clock cycle within the 10 clock-cycle range and recursively increases its precision (*i.e.,* $\frac{1}{10} * depth$) until a 100% success rate (10 out of 10 attempts) is achieved. In fact, this algorithm proved to be quite effective, locating the optimal parameters when attacking a `while(a)` loop in less than 59 minutes. Indeed, the algorithm achieved 7,031 successful glitches out of 36,869 in its search for when using `val != 0` as the comparator. When applied to a `while(a!=0xD3B9AEC6)` loop (*i.e.,* numbers with large Hamming distance), the algorithm converged in 16 minutes with 901 successful glitches.

C. Multi-glitch Attacks

Previous work has proposed implementing redundant checks to thwart glitching, which is based on the assumption that successfully glitching multiple instructions is a significant technical barrier for attackers [15], [76]. Indeed, multi-glitches are significantly more difficult in practice and, in some instances, can be impossible due to physical constraints. For example, the time required to recharge a capacitor could be greater than the time needed for the two glitches, which would prohibit EM or voltage glitching. Moreover, many systems have internal clocks, which thwart clock glitching, leaving these more-bounded glitching techniques as the only options in practice. We constructed an experiment to find the upper bound on the effectiveness of triggering an *identical* glitch twice in a row (*i.e.,* the ideal condition for an attacker as the same tuning parameters should work for both glitches) using clock glitching. We used the same comparisons that we used in our single glitch scenarios, but now with the trigger being reset, triggered, and a second glitch inserted (*i.e.,* two identical loops back-to-back). We recorded the number of successful partial glitches (*i.e.,* the first glitch was successful but the second was not) as well successful multi-glitches (*i.e.,* both glitches worked and the execution skipped both branch conditions). The results from these experiments can be seen in Table II.

TABLE II: The number of successful partial and multi-glitch attacks against three different branch guards implemented as infinite `while` loops

	while(!a)		while(a)		while(a!=0xD3B9AEC6)	
Cycle	**Partial**	**Full**	**Partial**	**Full**	**Partial**	**Full**
0	77	12	83	24	23	7
1	20	2	19	-	2	-
2	2	-	1	-	-	-
3	124	87	-	-	-	-
4	326	211	1	-	-	-
5	166	36	30	2	47	36
6	161	17	49	2	136	99
7	167	22	146	25	116	60
Total	1043	387	329	53	324	202
Total (%)	1.330%	0.494%	0.420%	0.068%	0.413%	0.258%

It is clear that multi-glitching is significantly more difficult in practice than a single glitch. The partial glitch success rates (*i.e.,* only the first glitch succeeded) are similar to those in our previous experiments: 1.330221%, 0.419600%, and 0.413223%, while the multi-glitch success rates (*i.e.,* the second glitch was also successful) were significantly lower: 0.493572%, 0.067595%, and 0.257627% respectively. Requiring a multi-glitch reduced the probability of a successful glitch by factors of $6\times$, `while(!a)`, $3\times$, `while(a)`, and $1.6\times$, `while(a!=0xD3B9AEC6)`. While these results may seem higher than previous work would indicate, this experiment was constructed to present the *best case* scenario for a multi-glitch. In practice, these factors would be significantly higher, since the attacker would not have 2 perfect triggers, the comparisons would likely not be identical, and there are numerous physical limitations to generating multiple glitches in rapid succession.

The large gap between partial glitches and successful multi-glitches is particularly interesting. This discrepancy leaves the potential to not only make glitching more difficult but to *detect* a glitching attempt, as a partial glitch introduces a logical impossibility, but would not skip the instrumented checks.

D. Long Glitch Attacks

While the multi-glitch results are encouraging, clock glitching permits an even more powerful attack. Specifically, an

attacker can inject a glitch at *every* clock cycle corrupting multiple contiguous instructions. Thus, we also tested the efficacy of a *long glitch* attack (*i.e.,* a glitch that is inserted for multiple clock cycles). In this experiment, we started by glitching 10 contiguous clock cycles (*i.e.,* the minimum number of clock cycles the two loops could possibly be completed in), and varied the clock cycles up to 20. For each number of repeated clock cycles, we varied the width and offset of the glitch in the same way as our previous experiments (9,801 glitching attempts per clock cycle range).

Despite the potential power of this attack, we observed mixed results (see Table III). The condition that was previously the most vulnerable, `while(!a)` faired much better against this attack, with far fewer successful glitches observed. We hypothesize that most successful glitching parameters, which disproportionately affect clock cycle 4 (*i.e.,* the compare instruction), are simultaneously corrupting the instructions before the comparator instructions and satisfying the exit condition. In the multi-glitch case the register would have contained 0, but in the long glitch case, it is likely that the subsequent load was also glitched, disrupting the ideal conditions for the previously observed single-clock-cycle attacks. Conversely, the `while(a)` case appeared to be significantly *more* susceptible to long glitch attacks, with over a $10\times$ increase in the success rate (*i.e.,* from 0.068% to 0.7%). We hypothesize that glitching so many load instructions could cause the various load instructions to fail, which would write 0 into the register and satisfying the exit condition. The higher number of success between 10 and 12 cycle glitches appears to support this claim, as after 12 clock cycles, the glitch would start to affect the compare and branch instructions of the second loop.

The lack of successes for the `while(a!=0xD3B9AEC6)` case coincides with our hypothesis that a glitch which simply changes the value in the register is unlikely to succeed. It appears that successful glitches against this case are corrupting the comparison instruction, the branch instruction, or the actual value loaded. In a multi-glitch scenario, the targeted glitch was affecting the same clock cycle both times, against identical code (*e.g.,* a branch condition). However, in the long glitch case, there are other instructions in the way that will also get glitched, making it exceedingly unlikely that *both* of the compare and branch instructions would be bypassed without irrecoverable corruption.

TABLE III: The number of successful long glitches against three unique branch guards implemented as two subsequent `while` loops, obtained by attempting all glitch offsets, widths, and number of clock cycles using a powerful clock glitch

Cycles	while(!a)	while(a)	while(a!=0xD3B9AEC6)
0-10	20	96	35
0-11	19	140	20
0-12	6	92	8
0-13	7	55	6
0-14	9	66	8
0-15	6	74	7
0-16	6	54	4
0-17	7	62	4
0-18	9	50	6
0-19	9	46	5
0-20	11	52	4
Total	109	787	107
Total (%)	0.101%	0.730%	0.0992%

VI. Glitching Defenses

While many glitching defenses have been proposed, few have been implemented, and we are unaware of any tool for generally applying these techniques. Thus, we present GlitchResistor, the first automated, open-source tool for implementing glitching defenses. GlitchResistor was implemented using the LLVM Project to modify both the source and compiled code (Clang and LLVM, respectively). This enables GlitchResistor to support multiple architectures with relatively low overhead. Indeed, many of the defenses *must* be implemented as a compiler pass, since implementing them in source code would result in the compiler optimizing them away (*i.e.,* because they appear as logically impossible or dead code). In this work, we only focused on the ARM architecture, specifically the STM32 microcontroller, due to its proliferation in embedded systems, its development support, and the supporting glitching frameworks [54]. However, our defenses work, without modification, on any architecture that is supported by LLVM (*e.g.,* MIPS, PowerPC, and RISC-V)

In general, software-based glitching defenses can be categorized into three broad categories: constant diversification, redundancy, and random timing.

A. Constant Diversification

Ideally, GlitchResistor would ensure that the set of enumerations (ENUMs) and return values would have a maximum, minimum pairwise Hamming distance (*i.e.,* the minimum Hamming distance between *all* of the values would be maximized) to minimize the chance of bit flips modifying a value into a different valid value. However, this is unfortunately an open coding theory problem in the general case, *i.e.,* $A(n, d)$ [46]. Thus, GlitchResistor instead leverages Reed-Solomon codes to generate values with large pairwise Hamming distances. In theory, this implementation can generate codes such that the minimum pairwise Hamming distance is $b - \lceil log_2(c) \rceil$ where b is the size of the value in bits and c is the number of values being generated. However, we used a more general purpose open-source implementation [45], which provides a flexible, fast computation of Reed-Solomon error codes. Our current implementation is configured with a message size of two bytes (*i.e.,* up to 2^{16} unique values in a set) and an ECC length equal to the size of values being generated (*e.g.,* 4 bytes for a typical ENUM). GlitchResistor then generates a message for each number $[1, count]$, where $count$ is the number of ENUMs in a particular definition, and uses the generated ECC as the new value in the program code ensuring a minimum pairwise Hamming distance of 8.

a) ENUM Rewriter: The ENUM Rewriter is the only defense implemented as a clang source code rewriter tool. This is because in the LLVM intermediate representation (IR), used by a compiler pass, ENUMs will be replaced by

corresponding constant values, and it is hard to detect which constant is the result of an ENUM expansion. Consequently, it is hard to replace ENUMs using a compiler pass in a sound manner. GLITCHRESISTOR first parses the abstract syntax tree (AST) of all the source and header files to identify ENUM declarations that have *all* of their values uninitialized. Then, for each of the uninitialized ENUM declarations, a set of Reed-Solomon codes is generated, and used as the declarations. GLITCHRESISTOR does not modify partially or fully initialized ENUM declarations, as they could represent certain expected values, and changing the values might affect the functionality of the firmware. Even for fully uninitialized enumerators, there could be cases where a programmer might assume default values for ENUMs (*i.e.,* starting with 0), as defined by the C standard [25]. *In such cases, without using a sound interval analysis [47], ENUM Rewriter might break the program's functionality.* To handle this, we provide an option in our implementation that will disable ENUM Rewriter. Developers could use this option if they made any assumptions about the default values for ENUMs in the target codebase.

b) Non-trivial Return Codes: GLITCHRESISTOR finds all of the functions that *only* return constant values using the LLVM *ModulePass*. For such functions, GLITCHRESISTOR examines how its callers use the return values. When they are *exclusively* used directly in branches (*i.e.,* compared to a constant) GLITCHRESISTOR replaces the return value and the constant that it is compared to with the hard-to-glitch values from our Reed-Solomon implementation. Our decision to only instrument functions that return constants reflects the fundamental difficulty in calculating all of the possible computed return values. Instrumentation that deals with such corner cases would be significantly more intrusive, and likely unsound. Our decision to only instrument return values that are used directly in branches could be relaxed, though only to a certain extent. If the instrumented constant is stored in an aliased memory location, significantly more heavyweight instrumentation would be required to dynamically track the value and update all of the references appropriately. Despite these minor limitations, our return code protection instruments a reasonable number of functions in practice (*i.e.,* 24 out of 312 *total* functions in our evaluated firmware).

B. Redundancy

GLITCHRESISTOR's redundancy defenses are implemented as an LLVM compiler pass that replicates existing code to ensure that no single-glitch attack will be capable of corrupting the execution. We ensure that code added for redundancy is *not* optimized out by other compiler passes by marking the inserted `load` and `store` instructions as volatile. These checks are capable of detecting glitches, as the injected check will never be false under normal operating conditions. Others have proposed and tested simple instruction duplication [18], concluding that instruction duplication alone is likely not a cure-all solution; hence the multi-pronged approach.

a) Data Integrity: GLITCHRESISTOR's data integrity protection is implemented by performing a *ModulePass*, which locates any global variables that were marked as *sensitive* by the developer (*e.g.,* by listing them in a configuration file). Once identified, these sensitive variables are replicated, and a second variable, which is used for verification, is allocated in a separate region of memory to ensure that it is not physically co-located with the initial variable. When a sensitive variable is written to memory, it is inverted (*i.e.,* `xor`ed with $\neg 0$ of the appropriate size), and this integrity value is stored in the complementary integrity variable. Then, when the value is later read from memory, both the original variable and the integrity value are read from memory and the operation will continue *if and only if* $var \oplus varIntegrity == \neg 0$, otherwise a glitch detection function will be called.

b) Branches and Loops: GLITCHRESISTOR implements two *FunctionPass* transformations to replicate conditional branch conditions. The first replicates the *true* condition for *every* conditional branch in the control-flow graph (CFG). When replicating the branch condition, GLITCHRESISTOR also replicates any instructions that are needed to calculate the comparison (*e.g.,* loading a value from memory, mutating it, and comparing it to an immediate). However, not every instruction can be replicated. For example, volatile variables, function calls, and LLVM `PHINodes` cannot be replicated because they may have adverse side-effects, or are likely to change between checks. This redundant comparison is computed to be the opposite of the initial branch condition (*e.g.,* `if (a == 5)` would become `if (¬a == ¬5)`), which ensures that the same bit flips repeated twice would not be able to bypass both checks. This defense assumes that security-critical operations are typically guarded by a conditional branch and that the default, *false*, branch is not as important to protect, as it will be taken most of the time. However, this assumption does not hold with loops. Thus, GLITCHRESISTOR performs a second pass to add the same redundant instrumentation to the *false* branch of loop guards.

c) Detection Reaction: GLITCHRESISTOR does not dictate an action to be taken when a glitch is detected, but instead provides a function that is trivially implemented by the developer. In fact, the specific reaction to a detected glitching attempt is *necessarily* application specific. For example, on a gaming system, it may be sufficient to simply report the attempt or disable updates, whereas a critical military system may want to react more assertively by completely destroying the data or device.

1) Random Timing: GLITCHRESISTOR currently injects randomness in the execution by injecting a random busy loop at the end of each basic block. The current implementation is a simple linear congruential generator (LCG) with the input parameters used by `glibc`, and each invocation executes between 0 and 10 no-operation (NOP) instructions. To ensure that any observable trigger is necessarily before the random function, the delay function is injected at the end of every basic block that ends in a `SwitchInst` or `BranchInst` (*i.e.,* right before a branch). This code injection was implemented as an LLVM *FunctionPass*. Functions can be easily omitted when the module is configured in *opt-out* mode or included

when it is configured in *opt-in* mode. Our seed is incremented, and written to flash, during the first invocation of the function (on our STM32 board, this was implemented in 10 lines of portable `C` code). GLITCHRESISTOR modifies the state of the random function immediately after the board boots (even before the board initializes) and writes the new seed to non-volatile memory to thwart repeated attempts against the same seed. This initialization code is also instrumented by the other defenses, which are capable of detecting glitching attempts.

VII. EVALUATION OF DEFENSES

GLITCHRESISTOR was both developed and evaluated on real hardware, using the STM32 suite of embedded devices. Two research questions arise with respect to defenses:

RQ6 How much overhead, both size and run-time, is incurred when using each GLITCHRESISTOR defense?

RQ7 How effective are the various GLITCHRESISTOR defenses at both mitigating and detecting glitching attacks?

A. Overhead

To evaluate the overhead imposed by GLITCHRESISTOR we first built a simple, indicative firmware using the STM32CubeMX code generator. This firmware initializes the board, and then loops forever, reading the number of ticks (*i.e.,* milliseconds) since the board was booted and printing out performance information after every loop iteration using the universal asynchronous receiver-transmitter (UART) interface. The variable that is used to store the tick counter was marked as a sensitive variable, and two functions that use ENUMs and constant return values are used to check the tick value. The firmware will call a *success* function if the tick value is ever equal to 0, which was designed to be impossible.

The specific board that we used in this experiment was an STM Nulceo 64 with an ARM Cortex-M4 (STM32F303RE)[2]. The default project, configured to be built with a `Makefile` was easily augmented to be built with LLVM and the appropriate GLITCHRESISTOR modules using a patching script that is provided with GLITCHRESISTOR. To ensure that there was no bias in the evaluation, we only measure the boot time of the system, as this code was provided by the CubeMX suite, and is used in numerous real systems. Moreover, the most security-critical code on embedded systems (*i.e.,* when GLITCHRESISTOR would provide the most value) is typically the bootloader. Each firmware was built using the default `-Og` optimization, which provides a *worst case* size. Eventually, we want to use existing static analysis techniques [40], [39] to further reduce the regions of code that need to be instrumented.

1) Run-time: To evaluate the boot process in a chip-agnostic way, we use the number of CPU cycles as our metric for comparison. This was done by enabling the data watchpoint and trace unit (DWT) on the board, and then reading the CPU once when the board is reset, and again after the hardware abstraction layer (HAL) and board had completely initialized.

[2]This is different from our glitching examples, because this board is more readily available and requires no special hardware to test with.

TABLE IV: Time overhead imposed by each defense on the boot time of a standard STM32 firmware image (clock cycles)

Defense	Clock Cycles (Avg.)	% Increase	Constant	% Adjusted
None	1736	0.00%	0	0.00%
Branches	1933	11.35%	0	11.35%
Delay	184388	10521.45%	177849	276.69%
Integrity	1737	0.06%	0	0.06%
Loops	1737	0.06%	0	0.06%
Returns	1739	0.17%	0	0.17%
All\Delay	2082	19.93%	0	19.93%
All	184761	10542.93%	177993	289.88%

Since our board is doing relatively simple operations, it only takes 1,736 clock cycles to boot in the un-instrumented case. We evaluated each defense independently, as they can be used *à la carte*. The results are shown in Table IV.

Injecting delays incurs a substantial constant overhead, as it must both read and write from flash memory the first time that it is called to update the seed to ensure that the pseudo-random number generator (PRNG) is unpredictable at every boot. When this constant overhead is accounted for, instrumenting *every* basic block in the boot process incurs a 277% overhead. However, in practice, a developer may want to use this particular feature in an "opt-in" way, such that it will only be applied to optionally annotated functions. Without the delay defenses enabled on every basic block, the run-time overhead incurred, in terms of clock cycles, is less than 20%. Nevertheless, both of these overheads are likely acceptable in practice to protect the critical code regions in a deployed embedded system.

2) Size: Since most embedded systems have strict constraints on their size, weight, and power (SWaP), we also enumerate how much additional code is inserted by GLITCHRESISTOR. Table V depicts the various code segments that are affected by each defense in GLITCHRESISTOR. Again, injecting a delay into *every* basic block incurs the largest overhead (13%). Meanwhile, the other defenses only combine for a 15% increase in size, most of which is in the `.text` segment. While modifying constant values (*i.e.,* returns and ENUMs) should, in theory, be "free," we actually see that they increase the size of the binary slightly because the transformed values are all necessarily four bytes, while smaller values can be encoded as a single byte. While these overheads may seem large after an initial glance, it is a small price to pay for the protection provided.

TABLE V: Size overhead imposed by each individual defense on a standard STM32 firmware built using CubeMX (bytes)

Defense	text	text (%)	data	data (%)	bss	bss (%)	total	total (%)
None	6456		120		1728		8304	
Branches	6956	7.74%	120	0.00%	1728	0.00%	8804	6.02%
Delay	7512	16.36%	128	6.67%	1768	2.31%	9408	13.29%
Integrity	6840	5.95%	124	3.33%	1732	0.23%	8696	4.72%
Loops	6840	5.95%	124	3.33%	1732	0.23%	8696	4.72%
Returns	6460	0.06%	120	0.00%	1728	0.00%	8308	0.05%
All\Delay	7700	19.27%	124	3.33%	1732	0.23%	9556	15.08%
All	9144	41.64%	132	10.00%	1768	2.31%	11044	33.00%

B. Effectiveness of Defenses

When testing these defenses against real glitches, we created both a *worst case* and *best case* scenario. In both cases we marked our variables as `volatile`, which hinders the effectiveness of the defenses (*i.e.,* they should perform *better* in practice). This is because the `volatile` variable cannot be read twice, which means that if the value were glitched successfully during the first load it would pass all checks. Thus, this analysis should provide a reasonable lower-bound for the effectiveness of these defenses in practice (*i.e.,* their ability to protect *any* code). Similarly, in both experiments, we attempted three different attack scenarios: a single glitch attack, where the clock cycle being glitched was varied (between 0 and 10); a long glitch attack, where the number of clock cycles being glitched was varied (between 10 and 100 at increments of 10); and a windowed long glitch attack, where the number of clock cycles was fixed at 10 (the best case in our previous experiment), but the initial clock cycle was varied (between 0 and 10 at increments of 10). All of the experiments had a *perfect* trigger, as before. These attacks are *far more powerful* than what an attacker would have access to on a real system, but, again, were constructed to provide a lower bound for the efficacy of the defenses.

1) `while(!a)` (worst case): The `while(!a)` condition was the most vulnerable against single-glitch attacks, and was thus chosen as our *worst case* scenario. As in Section V, we glitched the infinite loop, attempting to break out of it with the various defenses compiled into the code. While it should be theoretically impossible to defeat these defenses with a single glitch, the `volatile` variable leaves the possibility of successful glitching the register value and satisfy both conditional branches. The results from the three glitching attacks against this code are shown in Table VI.

The defenses turned out to be highly effective against the single-glitch attack, with success rates plummeting to less than 0.01%. Moreover, the detection rate is remarkably high (over 98%) both with and without the randomization defense enabled. This result is somewhat unsurprising, as these defenses were specifically constructed to ensure that no single incorrect branch would result in a compromised system [16]. However, the detection rates are especially encouraging with respect to real-world use cases for these defenses. The results are similarly positive against the more powerful long glitch attacks, with all of the defenses touting detection rates above 79%. It appears that the 10 cycle, a windowed glitch is far more effective against systems that do not have randomization enabled, since the more targeted window produces fewer detectable side effects. However, with randomization enabled, this attack performs slightly worse than the *longer* long glitch attack, likely due to the fact this shorter glitch window is more likely to corrupt a branch condition in the random function, which would be detected. On the contrary, since the long glitch attack will affect *every* clock cycle after the trigger, it can also glitch *all* of the detection code that it may touch.

TABLE VI: Successful glitches and detections against an infinite loop and a branch condition with GLITCHRESISTOR defenses. Successes(%): $\frac{suc.}{tot.}$, Detections(%): $\frac{det.}{det.+suc.}$

		`while(!a)`		`if(a==SUCCESS)`	
		All	All\Delay	All	All\Delay
Single	Total	107,811		107,811	
	Successes	10 (0.00928%)	4 (0.00371%)	-	1 (0.000928%)
	Detections	653 (98.4%)	1,032 (99.6%)	351 (100%)	95 (95.4%)
Long	Total	98,010		98,010	
	Successes	258 (0.263%)	262 (0.267%)	3 (0.00306%)	44 (0.0449%)
	Detections	981 (79.2%)	649 (71.2%)	1,143 (99.7%)	274 (86.2%)
10 Cycles	Total	107,811		107,811	
	Successes	227 (0.211%)	1,281 (1.188%)	10 (0.00557%)	2 (0.00186)
	Detections	1,858 (89.1%)	992 (43.6%)	2,019 (99.7%)	1016 (99.8%)

2) `if(a == SUCCESS)` (best case): In real code, infinite while loops are unlikely to guard security-critical code. Thus, to provide a more fair evaluation of the proposed defenses, we also attempted the three attacks against a simple `if` statement that is more indicative of how programmers write code. To ensure that all of the proposed defenses would be used, and to use the most resilient branch condition from Section V-C, we created an uninitialized `enum` variable: `SUCCESS`, which was initialized to `enum FAILURE`. This scenario should be the *best case* for the defenses (modulo the `volatile` variable), as the window for the a successful glitch is now quite narrow (*i.e.,* 8 clock cycles). The same attacks that from Section VII-B1 where used against this `if` statement; the results are shown in Table VI.

Indeed, the real power of these software-only defenses is exhibited in this case — only *one* single glitch attack was successful, with detection rates above (95%). The effectiveness of the long glitch attacks was similarly diminished. With all of the defenses enabled, the best attack was only able to achieve a 0.00557% success rate, with over 2,000 detections (a 99.7%) detection rate. Without the randomization defense enabled, the best attack was able to achieve a success rate of 0.0449%, with an 86.2% detection rate. While this experiment was constructed to be the best case scenario for the defenses, it is certainly not a corner case in real world code, demonstrating some real promise for these types of software-only defense against glitching in practice.

VIII. Related Work

In this section, we focus on work related to glitching defenses. Most of the hardware-based approaches are specific to a fault type. They require a precise sensitivity model [27], [84], [80], which is non-obvious for certain fault types such as those induced by an electromagnetic pulse. Recent work by Yuce *et al.* [85] shows that most of the hardware-based defenses are ineffective in the presence of multi-fault glitches (*e.g.,* voltage and EM). In this work, we focus on software-based defenses and develop a generic instrumentation technique that defends arbitrary software against various faults. Previous

TABLE VII: Comparison of GLITCHRESISTOR with existing software-based glitching defenses. Each technique is evaluated by checking whether the desirable properties are present (✓) or absent (✗).

Software-Based Defenses	Generic	Extensible	Backward Compatible	Defense Techniques			
				Constant Diversification	Data Integrity	Control flow Hardening	Random Delay
Data Encoding [37], [14]	✓	✗	✗	✓	✓	✗	✗
CAMFAS [17]	✗	✗	✗	✗	✗	✓	✗
Loop Hardening [60]	✓	✗	✓	✗	✗	✓	✗
IIR [58]	✗	✗	✓	✗	✓	✓	✗
CountCompile [11]	✓	✗	✗	✗	✓	✓	✗
CountC [36]	✓	✗	✗	✗	✗	✓	✗
SWIFT [63]	✓	✗	✓	✗	✗	✓	✗
CFCSS [55]	✓	✗	✗	✗	✗	✓	✗
GLITCHRESISTOR	✓	✓	✓	✓	✓	✓	✓

work [49], proposed and evaluated two software-only defenses (one which replicates instructions for redundancy [50] and the other which detects glitches [10]) for `bl` and `ldr` on ARM systems against electromagnetic fault injection (EMFI) glitching. These defenses are quite similar to our techniques for redundancy, and had similar success when they were evaluated. However, they noted that the countermeasure needed to be extended to a larger set of instructions and architectures, which GLITCHRESISTOR does by leveraging LLVM. Recent work [15] independently implemented, and evaluated branch duplication techniques in the context of spurious bit flips due to hardware malfunctions. Similar work [36] proposed a CFI method which implements a counter to detect if two more C source lines have been "skipped." However, this defense is especially heavyweight since it injects code after every instruction and it does not account for the possibility of a multi-glitch. Another work, CAMFAS [17], that uses SIMD [28] to replicate almost all instructions to detect fault attacks also suffers from the problem of being cumbersome and requires special hardware.

Most of the existing techniques are either application specific (*e.g.,* AES) or not backward compatible (*i.e.,* require the code of an entire program to be changed). On the other hand, GLITCHRESISTOR is generic, can be applied to any code, is backward compatible, and can be applied to selected program regions (*e.g.,* certain sensitive functions). GLITCHRESISTOR is based on the LLVM framework and is easily extended with other defenses. Table VII shows various software-based defenses in comparison with GLITCHRESISTOR demonstrating its holistic approach for defending against glitching.

Others have proposed a hybrid software and hardware approach where functions can be protected by inserting an `assert` function in the source, which will be updated with an LLVM pass to confer with the hardware and verify the "signature" of the function being executed [80]. GLITCHRESISTOR is differentiated by its fully-automated instrumentation and lack of mandatory source-code annotations.

Emulating glitching attacks has also been done previously. For example, one system implemented a fault injection emulator in the context of writing fault-tolerant code, but did not examine malicious glitching attacks [31]. Others similarly implemented a QEMU-based fault injection emulator [21] and glitch simulator [75] have been created to evaluate fault-tolerant techniques, both of which achieved mixed results. Nevertheless, since glitching is a physical phenomena, none of these emulators can adequately provide the realism of our real-world evaluation.

Previous work [82] presented comprehensive suggestions for source code modifications to make code glitch resistant, which our defenses are based on. Similarly, more recent work [81] advocated that "software mitigations like execution flow control, redundancy or random delays should be implemented" in embedded firmware. However, GLITCHRESISTOR is the first open-source framework for experimenting with various defenses and to test these defenses against attacks on *real hardware*, grounding our results and providing a more realistic view of their practical efficacy.

IX. CONCLUSION

In this work, we present GLITCHRESISTOR, an automated, open-source software-only defense framework. Our emulated experiments confirm that bit-level corruption can "skip" control flow instructions in ARM with a high likelihood *in theory* (60% when flipping to 0 and 30% when flipping to 1). Our real-world experiments demonstrated that glitching can be highly effective when all of the variables are controlled (*e.g.,* 100% success rate), and that the values being compared affect the *glitchability* of a particular branch instruction (*e.g.,* `while(!a)` was 2× more susceptible to glitching than `while(a)`). Moreover, we provide insights into *how* the control flow instructions are being skipped (*e.g.,* the register data being corrupted *versus* the execution being corrupted). We also demonstrate the complexity involved with multi-glitch attacks, whose difficultly is the basis of many proposed defenses. Finally, we show that GLITCHRESISTOR, with it's various software-only glitching defenses are capable of completely eliminating single-glitch attacks *in practice* and can minimize the likelihood of a successful multi-glitch attack , while detecting failed glitching attempts at a high rate.

ACKNOWLEDGEMENTS

We would like to acknowledge Fabian Monrose for his invaluable feedback on this work, Timothy Sherwood for his unwavering support, and our reviewers that helped to focus and strengthen this work through their comments. Similarly, we would like to thank Colin O'Flynn and the various experts at IBM Research and Riscure that we talked with for taking the time to help us construct realistic attacks and defenses.

This material is based upon work supported by the Office of Naval Research under Award No. N00014-17-1-2011, by the Department of Homeland Security under Award No. FA8750-19-2-0005, and by the IBM Ph.D. Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research, the Department of Homeland Security, or IBM Research.

REFERENCES

[1] "American fuzzy lop," http://lcamtuf.coredump.cx/afl/.

[2] A. ARM7TDMI, "Technical reference manual," *Advanced RISC Machines Ltd.,(15 May 2003)*.

[3] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 12, pp. 1295–1308, 2006.

[4] 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 *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2011, pp. 105–114.

[5] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[6] G. Barbu, G. Duc, and P. Hoogvorst, "Java card operand stack: fault attacks, combined attacks and countermeasures," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2011, pp. 297–313.

[7] G. Barbu, H. Thiebeauld, and V. Guerin, "Attacks on java card 3.0 combining fault and logical attacks," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2010, pp. 148–163.

[8] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low voltage fault attacks on the rsa cryptosystem," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2009, pp. 23–31.

[9] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi, "Low voltage fault attacks to aes," in *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2010, pp. 7–12.

[10] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented aes: effectiveness and cost," in *Proceedings of the 5th Workshop on Embedded Systems Security*. ACM, 2010, p. 7.

[11] T. Barry, D. Couroussé, and B. Robisson, "Compilation of a countermeasure against instruction-skip fault attacks," in *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, 2016, pp. 1–6.

[12] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet, "Combined software and hardware attacks on the java card control flow," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2011, pp. 283–296.

[13] C. Bozzato, R. Focardi, and F. Palmarini, "Shaping the glitch: Optimizing voltage fault injection attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 199–224, 2019.

[14] J. Breier, X. Hou, and Y. Liu, "On evaluating fault resilient encoding schemes in software," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[15] C.-K. Chang, G. Li, and M. Erez, "Evaluating compiler ir-level selective instruction duplication with realistic hardware errors," in *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 2019, pp. 41–49.

[16] T. Chen, "Guarding against physical attacks: The xbox one story," https://www.platformsecuritysummit.com/2019/speaker/chen/, October 2019.

[17] Z. Chen, J. Shen, A. Nicolau, A. Veidenbaum, N. F. Ghalaty, and R. Cammarota, "Camfas: A compiler approach to mitigate fault attacks via enhanced simdization," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2017, pp. 57–64.

[18] L. Cojocar, K. Papagiannopoulos, and N. Timmers, "Instruction duplication: Leaky and not too fault-tolerant!" in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2017, pp. 160–179.

[19] A. Cui and R. Housley, "BADFET: Defeating modern secure boot using second-order pulsed electromagnetic fault injection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[20] M. Dadashi, L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Hardware-software integrated diagnosis for intermittent hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 363–374.

[21] F. de Aguiar Geissler, F. L. Kastensmidt, and J. E. P. Souza, "Soft error injection methodology based on qemu software platform," in *2014 15th Latin American Test Workshop-LATW*. IEEE, 2014, pp. 1–5.

[22] E. DeBusschere and M. McCambridge, "Modern game console exploitation," *Technical Report, Department of Computer Science, University of Arizona*, 2012.

[23] C. Dobraunig, M. Eichlseder, H. Gross, S. Mangard, F. Mendel, and R. Primas, "Statistical ineffective fault attacks on masked aes with fault countermeasures," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2018, pp. 315–342.

[24] J. W. Duran and S. Ntafos, "A report on random testing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 179–183.

[25] I. O. for Standardization, "Iso/iec 9899:tc3: Programming languages - c," 2007.

[26] A. Galauner, "Glitching the switch," https://media.ccc.de/v/c4.openchaos.2018.06.glitching-the-switch#t=82, June 2018.

[27] N. F. Ghalaty, A. Aysu, and P. Schaumont, "Analyzing and eliminating the causes of fault sensitivity analysis," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[28] R. J. Gove, K. Balmer, N. K. Ing-Simmons, and K. M. Guttag, "Multiprocessor reconfigurable in single instruction multiple data (simd) and multiple instruction multiple data (mimd) modes and method of operation," May 18 1993, uS Patent 5,212,777.

[29] J. Grand and J. Friday, "Advanced hardware hacking techniques," *DEFCON*, vol. 12, p. 59, 2004.

[30] J. Gratchoff, N. Timmers, A. Spruyt, and L. Chmielewski, "Proving the wild jungle jump," 2015.

[31] A. Höller, G. Macher, T. Rauter, J. Iber, and C. Kreiner, "A virtual fault injection framework for reliability-aware software development," in *2015 IEEE International Conference on Dependable Systems and Networks Workshops*. IEEE, 2015, pp. 69–74.

[32] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[33] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 361–372.

[34] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[35] T. Korak and M. Hoefler, "On the effects of clock and power supply tampering on two microcontroller platforms," in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2014, pp. 8–17.

[36] J.-F. Lalande, K. Heydemann, and P. Berthomé, "Software countermeasures for control flow integrity of smart card c codes," in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 200–218.

[37] M. M. Latif, R. Ramaseshan, and F. Mueller, "Soft error protection via fault-resilient data representations," North Carolina State University. Dept. of Computer Science, Tech. Rep., 2007.

[38] N. Lawson, "How the ps3 hypervisor was hacked," https://rdist.root.org/2010/01/27/how-the-ps3-hypervisor-was-hacked/, 2010.

[39] G. Li, Q. Lu, and K. Pattabiraman, "Fine-grained characterization of faults causing long latency crashes in programs," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 450–461.

[40] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 27–38.

[41] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[42] Y. Lu, "Attacking hardware aes with dfa," https://yifan.lu/images/2019/02/Attacking_Hardware_AES_with_DFA.pdf, Februrary 2019.

[43] ——, "Injecting software vulnerabilities with voltage glitching," *arXiv preprint arXiv:1903.08102*, 2019.

[44] Y. Lu and Davee, "Viva la vita vida: Hacking the most secure handheld console," https://media.ccc.de/v/35c3-9364-viva_la_vita_vida, December 2018.

[45] M. Lubinets, "Reed solomon bch encoder and decoder," https://github.com/mersinvald/Reed-Solomon, Februrary 2016.

[46] M. Milshtein, "A new two-error-correcting binary code of length 16," *Cryptography and Communications*, pp. 1–5, 2019.

[47] R. E. Moore, R. B. Kearfott, and M. Cloud, "Introduction to interval analysis," 2009.

[48] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2013, pp. 77–88.

[49] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz, "Experimental evaluation of two software countermeasures against fault attacks," in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2014, pp. 112–117.

[50] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 145–156, 2014.

[51] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[52] A. Q. Nguyen and H. V. Dang, "Unicorn: Next generation cpu emulator framework," in *Proceedings of the 2015 Blackhat USA conference*, 2015.

[53] C. O'Flynn, "Fault injection using crowbars on embedded systems." *IACR Cryptology ePrint Archive*, vol. 2016, p. 810, 2016.

[54] C. O'Flynn and Z. D. Chen, "Chipwhisperer: An open-source platform for hardware embedded security research," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2014, pp. 243–260.

[55] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[56] R. Omarouayache, J. Raoult, S. Jarrix, L. Chusseau, and P. Maurine, "Magnetic microprobe design for em fault attack," in *2013 International Symposium on Electromagnetic Compatibility*. IEEE, 2013, pp. 949–954.

[57] S. Ordas, L. Guillaume-Sage, and P. Maurine, "Electromagnetic fault injection: the curse of flip-flops," *Journal of Cryptographic Engineering*, vol. 7, no. 3, pp. 183–197, 2017.

[58] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont, "Lightweight fault attack resistance in software using intra-instruction redundancy," in *International Conference on Selected Areas in Cryptography*. Springer, 2016, pp. 231–244.

[59] F. Project, "The xbox 360 reset glitch hack," https://free60project.github.io/wiki/Reset_Glitch_Hack.html.

[60] J. Proy, K. Heydemann, A. Berzati, and A. Cohen, "Compiler-assisted loop hardening against fault attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, pp. 1–25, 2017.

[61] N. A. Quynh, "Capstone: Next generation disassembly framework," *Black Hat USA*, 2014.

[62] ——, "Keystone: Next generation assembler framework," *Black Hat USA*, 2016.

[63] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*. IEEE, 2005, pp. 243–254.

[64] L. Riviere, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage, "High precision fault injections on the instruction cache of ARMv7-M architectures," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015, pp. 62–67.

[65] F. Rodríguez, J. C. Campelo, and J. J. Serrano, "A watchdog processor architecture with minimal performance overhead," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2002, pp. 261–272.

[66] F. Rodríguez and J. J. Serrano, "Control flow error checking with isis," in *International Conference on Embedded Software and Systems*. Springer, 2005, pp. 659–670.

[67] T. Roth, "TrustZone-M(eh): Breaking ARMv8-M's security," in *The 36th Chaos Communication Congress (36C3)*, December 2019.

[68] G. T. H. G. Roussel-Tarbouriech, N. Menard, T. True *et al.*, "Methodically defeating nintendo switch security," *arXiv preprint arXiv:1905.07643*, 2019.

[69] J.-M. Schmidt and C. Herbst, "A practical fault attack on square and multiply," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2008, pp. 53–58.

[70] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, 2015.

[71] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2002, pp. 2–12.

[72] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: sanitizing for security," *IEEE Security and Privacy*, 2019.

[73] C. Spensky, "Analyzing and securing embedded systems," Ph.D. dissertation, UC Santa Barbara, 2020.

[74] A. Spruyt, "Building fault models for microcontrollers," *University of Amsterdam, Amsterdam, Tech. Rep*, pp. 2011–2012, 2012.

[75] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 404–409.

[76] N. Timmers and C. Mune, "Escalating privileges in linux using voltage fault injection," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2017, pp. 1–8.

[77] N. Timmers and A. Spruyt, "Bypassing secure boot using fault injection," *Black Hat Europe*, vol. 2016, 2016.

[78] N. Timmers, A. Spruyt, and M. Witteman, "Controlling PC on ARM using fault injection," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2016, pp. 25–35.

[79] J. G. Van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2011, pp. 91–99.

[80] M. Werner, E. Wenger, and S. Mangard, "Protecting the control flow of embedded processors against fault attacks," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2015, pp. 161–176.

[81] N. Wiersma and R. Pareja, "Safety!= security: On the resilience of asil-d certified microcontrollers against fault injection attacks," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2017, pp. 9–16.

[82] M. Witteman and M. Oostdijk, "Secure application programming in the presence of side channel attacks," in *RSA conference*, vol. 2008, 2008.

[83] A. G. Yanci, S. Pickles, and T. Arslan, "Detecting voltage glitch attacks on secure devices," in *2008 Bio-inspired, Learning and Intelligent Systems for Security*. IEEE, 2008, pp. 75–80.

[84] B. Yuce, N. F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. Schaumont, "Fame: Fault-attack aware microprocessor extensions for hardware fault detection and software fault response," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, 2016, pp. 1–8.

[85] B. Yuce, P. Schaumont, and M. Witteman, "Fault attacks on secure embedded software: Threats, design, and evaluation," *Journal of Hardware and Systems Security*, vol. 2, no. 2, pp. 111–130, 2018.