



A Fault Injection Test Bench for Evaluating Secure Boot Resilience in Open Instruction Set Processors

Naveen Grant^{1*}, Kavita Zulu², Anna Harkness³

¹Department of Electrical Electronics, Mekong Graduate Institute of Electrical Electronics, Philippines

²Department of Electrical Electronics, Irrawaddy Research Institute of Electrical Electronics, Philippines

³Department of Electrical Electronics, Ural University Centre for Electrical Electronics, Taiwan

Article Info

Volume 1, Issue 1, 2026

doi: [10.62715/ijesd.2026.v1.i1.pp20-25](https://doi.org/10.62715/ijesd.2026.v1.i1.pp20-25)

Received: 18 November 2025

Accepted: 30 January 2026

Published: 06 April 2026

Abstract

Secure boot is the root of trust in most embedded devices, and it can be defeated without breaking any cryptography if a physical fault makes the processor skip or corrupt the instruction that acts on the verification result. Processors built on the open RISC-V instruction set are moving into products where such attacks are in scope, and they offer evaluators something closed cores do not: access, in many cases, to the register-transfer level design. This article proposes a test bench and an accompanying procedure for evaluating how well a secure boot implementation on a RISC-V target resists voltage, clock and electromagnetic fault injection. The bench combines a target board prepared for glitching, a low-cost injection platform, a triggering path derived from the boot sequence, and a host controller that sweeps injection parameters and classifies outcomes. The procedure runs in six phases, from threat model to countermeasure re-test, and separates the characterisation of the fault model on instrumented code from the campaign against the boot code itself. Recommendations are given for outcome classification, for reporting success rates with confidence intervals, and for linking physical results to simulation at the instruction and register-transfer levels. The bench and procedure are proposals built from published practice. They have not been validated experimentally in this article.

Keywords: Fault injection, Secure boot, RISC-V, Voltage glitching, Electromagnetic fault injection, Hardware security evaluation

© 2026 Naveen Grant et al. This is an open access article under the CC BY license (<https://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

1. Introduction

A secure boot chain starts from immutable code, checks the integrity and authenticity of the next stage before running it, and repeats the check at each hand-over. The architecture was described by [Arbaugh et al. \(1997\)](#) and is now assumed by platform firmware guidance such as NIST SP 800-193 ([Regenscheid, 2018](#)). Its security argument rests on the signature scheme. Its implementation rests on a handful of instructions: a comparison, a conditional branch, a return value. If one of those misbehaves at the right instant, an unsigned image runs.

* Corresponding author: Naveen Grant, Department of Electrical Electronics, Mekong Graduate Institute of Electrical Electronics, Philippines.
E-mail: naveen.grant@mekong.edu

© 2026 Naveen Grant et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Fault injection makes instructions misbehave. Since Boneh *et al.* (1997) showed that a single computational fault can expose a private key, the technique has grown from a threat to smart cards into a general tool against embedded software (Bar-El *et al.*, 2006; Barengi *et al.*, 2012). Published attacks on boot loaders have bypassed signature checks and read-out protection on commercial devices with voltage and electromagnetic pulses (Cui and Housley, 2017; Van den Herrewegen *et al.*, 2021).

Processors that implement the RISC-V instruction set (Waterman and Asanovic, 2019) are now used as security controllers, application cores and roots of trust. Their boot code needs the same scrutiny as any other, and evaluators need a repeatable way to apply it. This article proposes a test bench and a procedure for that purpose. It describes what to build, in what order to work and what to record. It reports no experimental campaign, and the bench has not been validated here.

2. Background

The design draws on three strands of published work: the physical means of injecting faults, the fault models observed on microcontrollers, and the particular features of open instruction set targets.

2.1. Injection techniques

Four techniques account for most practical attacks. Voltage glitching briefly lowers or raises the supply so that a critical path misses timing; the shape of the glitch matters as well as its width and delay, as Bozzato *et al.* (2019) showed with arbitrary waveforms. Clock glitching inserts a shortened cycle and applies where the attacker controls an external clock. Electromagnetic fault injection induces currents with a pulsed coil near the package and adds spatial selectivity without decapsulation (Moro *et al.*, 2013). Optical injection with a laser or flash on an exposed die is the most precise and the most demanding in preparation (Skorobogatov and Anderson, 2003). Open hardware such as ChipWhisperer has brought the first three within reach of an ordinary laboratory (O'Flynn and Chen, 2014).

2.2. Fault models on software

The effect most often exploited is the skipped or replaced instruction. Moro *et al.* (2013) traced electromagnetic faults on a 32-bit microcontroller to corruption of the instruction as it is fetched, which at the software level looks like one instruction being replaced by another. More powerful effects are possible: Timmers *et al.* (2016) showed that a fault during a load can place attacker-controlled data in the program counter on a 32-bit Arm core. The survey by Yuce *et al.* (2018) connects such physical effects to their consequences for embedded software and to the design of countermeasures, among them instruction duplication schemes whose correctness against single skips has been formally verified (Moro *et al.*, 2014).

2.3. Open instruction set targets

RISC-V targets differ from closed cores in ways that affect an evaluation. Many are available as register-transfer level source, so a fault observed on silicon or on a field-programmable gate array can be compared with simulation of the same design. Laurent *et al.* (2019) used that access to show that faults in registers hidden from the programmer, such as pipeline and forwarding registers, produce behaviours that an instruction-level model does not predict. The instruction encoding matters too. Where the compressed extension is enabled, 16-bit and 32-bit instructions are mixed, and a corrupted word may be decoded as a different sequence altogether. Privilege is managed through machine mode and physical memory protection as defined in the privileged specification (Waterman *et al.*, 2021), and the configuration of those registers during boot is itself a target.

3. Architecture of the Proposed Bench

The bench has five elements. The first is the target. A board is prepared by removing decoupling capacitors from the core supply rail, inserting a shunt resistor for power measurement, and exposing the reset and clock lines. Where the processor is a soft core, a gate array board built for security evaluation serves the same purpose, with the reservation noted later that faults in programmable logic are not the faults of the final silicon. The second element is the injector: a crowbar or multiplexer circuit for voltage

glitches, a clock generator able to insert a partial cycle, and optionally a pulsed electromagnetic probe on a motorised stage.

The third element is the trigger, which fixes the time reference for each attempt. A dedicated output pin toggled by instrumented firmware is the cleanest reference and is used during characterisation. For production boot code the trigger has to come from outside: the release of reset, a pattern on a serial or flash bus, or a feature in the power trace recognised in real time. The fourth element is observation. The serial console, a status pin and, where it is open, the debug port tell the bench what the target did after each attempt. The fifth is the host controller, a script that sets the parameters, arms the injector, resets the target, collects the response and logs every attempt with its settings.

4. The Evaluation Procedure

The procedure has six phases. Table 1 lists them with their outputs. The ordering reflects a principle taken from evaluation practice: establish that faults can be induced, and of what kind, before spending time on the code that matters.

Table 1: Phases of the proposed evaluation procedure and the output of each		
Phase	Activity	Output
1. Threat model	State attacker access, equipment class and the assets the boot chain protects	Scope statement and success criteria
2. Bench calibration	Confirm glitch shape at the pin and trigger jitter with an oscilloscope	Calibration record
3. Fault characterisation	Sweep parameters over instrumented loops and register patterns	Parameter map and observed fault model
4. Timing localisation	Locate boot checks in time using traces and bus activity	Candidate injection windows
5. Boot campaign	Attack each window with the characterised parameters; classify outcomes	Per-window outcome counts and intervals
6. Countermeasure re-test	Repeat phase 5 with hardening enabled, including multi-fault attempts	Comparison report and residual risk

4.1. Phases 1 and 2: Scope and calibration

The threat model states whether the attacker has unlimited physical access, how many devices may be consumed, and which class of equipment is assumed. It also defines success. For secure boot the usual criteria are execution of an unauthenticated image, re-enabling of a locked debug interface, and disclosure of key material. Calibration then checks that the bench does what the script believes: the glitch is observed at the target pin and not at the generator, and the spread of the trigger delay is measured, because jitter larger than one clock period blurs every later result.

4.2. Phase 3: Characterising the fault model

Characterisation runs on firmware written for the purpose. A typical test program raises the trigger, executes a counted loop or a sequence of register moves with known results, and prints the final state. The controller sweeps glitch delay, width and, for electromagnetic injection, probe position and pulse amplitude. Each attempt falls into one of four classes: no effect, reset or mute, crash with an exception, or a wrong result with continued execution. The last class is the useful one. Inspecting the wrong results shows whether instructions are being skipped, operands corrupted or loads disturbed, and the parameter region in which this happens most reliably becomes the starting point for the campaign. Test collections designed for fault analysis can supply the instrumented code.

4.3. Phase 4: Locating the boot checks in time

Figure 1 pairs the stages of a generic boot chain with the operations in each that a fault might usefully disturb. Locating those operations in time is the task of the fourth phase. Reading the image from flash produces visible bus activity, the digest computation has a regular power signature whose length scales with the image, and the signature verification follows it. The windows of interest are short: the comparison of the computed and expected values, the branch on its result, and the write that records the verdict. Where source or a disassembly of the boot code is available, the windows can be predicted by cycle counting and then confirmed against the traces.

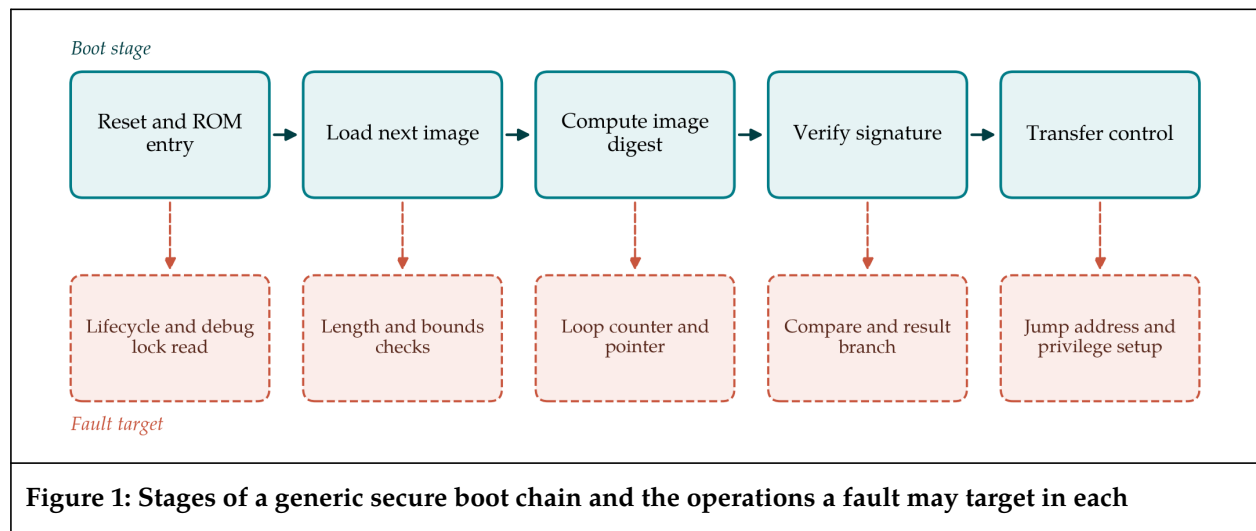


Figure 1: Stages of a generic secure boot chain and the operations a fault may target in each

4.4. Phase 5: The boot campaign

The campaign applies the characterised parameters to each candidate window with an unsigned or tampered image in place. The outcome classes are extended with the success criteria fixed in the first phase. Every attempt is logged with its parameters, since a single success needs to be reproducible to count. Results are reported per window as the number of successes among the attempts made, together with a confidence interval for the underlying rate. The score interval of [Wilson \(1927\)](#) behaves well for the small proportions typical of this work. A window with no success after many attempts supports only an upper bound on the success rate, and the report should say so in those terms.

4.5. Phase 6: Re-testing with countermeasures

Software hardening of boot code commonly includes repeating the verification, encoding the verdict in constants with a large Hamming distance in place of zero and one, checking that execution arrived by the expected path, and inserting random delays. Hardware adds glitch detectors and redundant logic. The final phase repeats the campaign with these measures enabled. Because duplicated checks can be defeated by two well-timed faults, the bench should support multiple glitches per boot. The comparison of success rates before and after hardening, under an unchanged injection set-up, is the main product of the evaluation.

5. Linking the Bench to Simulation

An open design allows the physical results to be explained and extended. At the instruction level, an emulator can replay the boot code while skipping or corrupting each instruction in turn, which lists every single fault that leads to a bypass under the assumed model and indicates where the bench should aim. At the register-transfer level the same exercise can be run on pipeline and control registers, which addresses the hidden-state effects that instruction-level models miss ([Laurent et al., 2019](#)). Agreement between the faults found physically and those the simulation predicts raises confidence in both. Disagreement is informative as well, since it shows the limits of the fault model in use.

6. Limitations and What Remains to Be Shown

The bench is a proposal and its usefulness is untested here. Several points would need to be established by applying it. Repeatability comes first: whether two laboratories following the procedure on the same target obtain comparable parameter maps and success rates. Transfer is a second question. A soft core on a gate array shares its logic design with the eventual chip and nothing of its physical layout, so results on one may say little about the other. Coverage is limited by the techniques included; laser injection, which the bench omits on grounds of cost, may reach faults the others do not. A negative result has limited meaning in any case, since the absence of a successful fault under a given budget of attempts and equipment is not evidence of immunity. Electromagnetic and high-voltage pulse equipment also brings electrical hazards that a laboratory must manage under its own safety rules.

7. Conclusion

Evaluating secure boot against fault injection requires modest equipment and a disciplined method more than it requires rare expertise. The bench proposed here assembles a prepared target, an affordable injector, a trigger path, observation channels and a logging controller, and the six-phase procedure moves from fault characterisation on known code to a measured campaign against the boot chain and its countermeasures. For RISC-V targets the availability of the design source allows physical results to be tied back to simulation, an opportunity that closed cores rarely afford. Whether the procedure yields consistent and transferable results is the question that experimental use must now answer.

Conflict of interest

To be declared by the author(s) before publication.

Funding

To be declared by the author(s) before publication.

References

- Arbaugh, W.A., Farber, D.J. and Smith, J.M. (1997). [A secure and reliable bootstrap architecture. *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 65-71.](#)
- Bar-El, H., Choukri, H., Naccache, D., Tunstall, M. and Whelan, C. (2006). [The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94\(2\): 370-382. <https://doi.org/10.1109/JPROC.2005.862424>](#)
- Barenghi, A., Breveglieri, L., Koren, I. and Naccache, D. (2012). [Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100\(11\): 3056-3076. <https://doi.org/10.1109/JPROC.2012.2188769>](#)
- Boneh, D., DeMillo, R.A. and Lipton, R.J. (1997). [On the importance of checking cryptographic protocols for faults. *Advances in Cryptology, EUROCRYPT 1997, Lecture Notes in Computer Science*, 1233: 37-51. \[https://doi.org/10.1007/3-540-69053-0_4\]\(https://doi.org/10.1007/3-540-69053-0_4\)](#)
- Bozzato, C., Focardi, R. and Palmarini, F. (2019). [Shaping the glitch: Optimizing voltage fault injection attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019\(2\): 199-224.](#)
- Cui, A. and Housley, R. (2017). [BADFET: Defeating modern secure boot using second-order pulsed electromagnetic fault injection. *Proceedings of the 11th USENIX Workshop on Offensive Technologies \(WOOT\)*.](#)
- Laurent, J., Beroulle, V., Deleuze, C. and Pebay-Peyroula, F. (2019). [Fault injection on hidden registers in a RISC-V Rocket processor and software countermeasures. *Proceedings of the Design, Automation and Test in Europe Conference \(DATE\)*, 252-255.](#)
- Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B. and Encrenaz, E. (2013). [Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography \(FDTC\)*, 77-88.](#)

- Moro, N., Heydemann, K., Encrenaz, E. and Robisson, B. (2014). [Formal verification of a software countermeasure against instruction skip attacks](#). *Journal of Cryptographic Engineering*, 4(3): 145-156.
- O'Flynn, C. and Chen, Z. (2014). [ChipWhisperer: An open-source platform for hardware embedded security research](#). *Constructive Side-Channel Analysis and Secure Design (COSADE), Lecture Notes in Computer Science*, 8622: 243-260.
- Regenscheid, A. (2018). [Platform Firmware Resiliency Guidelines](#). NIST Special Publication 800-193, National Institute of Standards and Technology, Gaithersburg, MD.
<https://doi.org/10.6028/NIST.SP.800-193>
- Skorobogatov, S.P. and Anderson, R.J. (2003). [Optical fault induction attacks](#). *Cryptographic Hardware and Embedded Systems, CHES 2002, Lecture Notes in Computer Science*, 2523: 2-12.
- Timmers, N., Spruyt, A. and Witteman, M. (2016). [Controlling PC on ARM using fault injection](#). *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 25-35.
- Van den Herrewegen, J., Oswald, D., Garcia, F.D. and Temeiza, Q. (2021). [Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis](#). *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1): 56-81.
- Waterman, A. and Asanovic, K. (eds.) (2019). [The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213](#). RISC-V Foundation.
- Waterman, A., Asanovic, K. and Hauser, J. (eds.) (2021). [The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203](#). RISC-V International.
- Wilson, E.B. (1927). [Probable inference, the law of succession, and statistical inference](#). *Journal of the American Statistical Association*, 22(158): 209-212.
- Yuce, B., Schaumont, P. and Witteman, M. (2018). [Fault attacks on secure embedded software: Threats, design, and evaluation](#). *Journal of Hardware and Systems Security*, 2(2): 111-130.
<https://doi.org/10.1007/s41635-018-0038-1>