High Precision Laser Fault Injection using Low-cost Components.

Martin S. Kelly
Information Security Group
Smart Card Centre
Royal Holloway, University of London
Egham, TW20 0EX
United Kingdom.
Email: Martin.Kelly.2014@live.rhul.ac.uk

Keith Mayes
Information Security Group
Smart Card Centre
Royal Holloway, University of London
Egham, TW20 0EX
United Kingdom.
Email: Keith.Mayes@rhul.ac.uk

***Abstract*—This paper demonstrates that it is possible to execute sophisticated and powerful fault injection attacks on micro-controllers using low-cost equipment and readily available components. Earlier work had implied that powerful lasers and high grade optics frequently used to execute such attacks were being underutilized and that attacks were equally effective when using low-power settings and imprecise focus.**

This work has exploited these earlier findings to develop a low-cost laser workstation capable of generating multiple discrete faults with timing accuracy capable of targeting consecutive instruction cycles. We have shown that the capabilities of this new device exceed those of the expensive laboratory equipment typically used in related work.

We describe a simplified fault model to categorize the effects of induced errors on running code and use it, along with the new device, to reevaluate the efficacy of different defensive coding techniques. This has enabled us to demonstrate an efficient hybrid defense that outperforms the individual defenses on our chosen target.

This approach enables device programmers to select an appropriate compromise between the extremes of undefended code and unusable overdefended code, to do so specifically for their chosen device and without the need for prohibitively expensive equipment. This work has particular relevance in the burgeoning IoT world where many small companies with limited budgets are deploying low-cost microprocessors in ever more security sensitive roles.

I. Introduction

In earlier work [1] we have shown that, given accurate synchronisation between instruction execution and laser pulse generation, the errors induced in a running microprocessor (μP) can be highly repeatable and that the dominant effect is the skipping (or misreading) of the instruction being fetched at the time of the pulse. Furthermore it was noted that these repeatable effects can be observed at modest power settings and with beam spot sizes that are readily achievable without the need for expensive precision optics, a result also noted by [2]. Limitations of our equipment at that time prevented us from exploring the effects of multiple laser pulses.

Solid state lasers are available with fast switching characteristics and offering sufficient power to compare with our previous findings (circa $2\,\mathrm{mJ}$ per pulse). The solid state lasers are not only much cheaper than the YAG laser cutter we previously used, but they also overcome its main operational drawback, namely the $20\,\mathrm{ms}$ recharge time between consecutive laser pulses. By using these fast laser diodes and suitable control circuitry we have generated pulses as short as $5\,\mathrm{ns}$ ($200\,\mathrm{MHz}$) with sufficient power to induce errors in our sample μPs. This enabled us to accurately synchronize laser pulses with each and any instruction the μP was executing. Results from this test rig compare favorably with our old equipment.

Section II provides a brief overview of the background and motivation for the work which led to development of the test rig. Section III describes the components used in the test rig. Section IV demonstrates the capabilities of the test rig by precisely controlling a series of conditional branch instructions and in Section V we describe our simplified fault model and characterize an extensive set of defensive coding techniques. Our conclusions are summarized in Section VI.

II. Background

The risks and consequences of errors in computation have been well understood for a long time [3], [4]. As a consequence it is common practice for software to perform sanity checks on its own data and calculations. This is commonly referred to as defensive programming. Trade organizations such as EMVCo issue their own guidelines to application developers to ensure the appropriate software defenses are utilized in a secured smart card application [5]–[7]. Unfortunately software defenses come at the cost of impaired performance and increased code volume. Ultimately a compromise has to be made between security, performance and cost.

With the exception of software bugs, errors occur in computational results as a result of a malfunction of the processing engine or corrupted data in memory. Deliberate induction of errors in a semiconductor devices can be archived through a variety of mechanisms. Momentary glitches affecting the μP's clock signal or supply voltage [8] can cause execution errors. Laser, Electro-Magnetic pulses or strong localized EM fields can be equally effective at inducing errors in μPs, [9]–[12].

The use of lasers to induce errors is a 'semi-invasive' attack. Here the chip's packaging must be removed to enable access to its surface but the chip itself is not modified. These

techniques date back to the mid 1960's and were used to simulate the effects of intense radiation on silicon devices [9]. The primary interest back then was system reliability in hostile environments. The use of a focused laser pulse as an attack technique was first brought to academic attention at the turn of this century [10].

Glitch attacks are 'non-invasive' in that they can be performed on the device in its unmodified packaging. The disadvantage of such non-invasive attacks is that the stimulus is typically applied to the whole (or significant fraction of) the target. Multiple sub-components can be affected and the resulting disruption can have many side effects throughout the chip giving the impression that induced errors are random in nature. Semi-invasive attacks can be focused upon significantly smaller active areas of the chip, producing localized effects. It has been demonstrated that these local effects are non-random and readily repeatable [1].

It is not always possible to induce errors using visible light via the top-side of a silicon chip. The metal tracks connecting the silicon structures frequently obscure vulnerable parts of the chip, and in high security devices, such as smart cards, metal layers are deliberately placed to shield the chip [13]. In these circumstances it is often possible to perform an attack from the backside of the chip [14]. At near infrared light wavelength ($1064\,\mathrm{nm}$) silicon is effectively transparent and this enables access to the chip's transistors behind any metal features on the topside. [2] has demonstrated the same localized and repeatable error effects on both AVR & ARM μPs using this technique.

More recently studies have focused on characterising the nature of the faults observed in a chip under attack. By looking at the Instruction Set Architecture (ISA) researchers have categorized faults as 'load value corruption' or 'instruction replacement' [15], [16]. These studies seek to understand the mechanism underlying the faults whereas our interest remains in the effects of the fault on subsequent computations. For our model it matters not whether the μP failed to perform an ADD instruction or if it added faulty data; either way the μP continues its computations with erroneous data. This simple fault model gives us an efficient mechanism for recognising faults and measuring the efficacy of defenses.

Previously [17] constructed a very low-cost fault injector using a flash gun and most significantly demonstrated its capabilities against differing μP architectures. Unfortunately this device suffers from a slow recharge time, limiting its application to single fault events. Single fault events are relatively easy to defend against in software.

To overcome sophisticated software defenses requires multiple closely timed error events. Our goal was to do precisely this, and to do so on a limited budget. Thereby finally dispelling the myth that localized semi-invasive optical fault attacks are difficult to perform and prohibitively expensive for attackers with a limited budget.

III. Equipment

We built our laser station using readily available low-cost components to demonstrate that the attack capabilities of such a system are practical and available to a wider range of attackers than would have been possible with the professional/commercial laser systems used in earlier work.

A. Components

The most readily available laser diodes offering power outputs in the 4W range and capable of being switched at up to $200\,\mathrm{MHz}$ are in the 400 ... $455\,\mathrm{nm}$ (violet ... blue) wavelength range. We chose a $455\,\mathrm{nm}$ laser diode salvaged from a high-intensity Nichia NUMB80 laser diode bank [18]. It offers up to $4.3\,\mathrm{W}$ of power and is capable of switching at $200\,\mathrm{MHz}$. Available on Ebay for less than $40 in 2018.

Driving the laser-diode at full power requires a circuit capable of switching a $3\,\mathrm{A}$ current without creating excessive spikes or reverse voltages which can be fatal to a diode. We used a dedicated laser controller device, the iCHaus iCHG 3A Laser switch [19], which is typically used for LiDAR and data transmission. It is conveniently available through distributors on an evaluation board for $70.

We used an FPGA to provide both the clock for the μP and the laser trigger pulses. We programmed this device to provide a number of high speed counters, giving us control of both the timing and duration of multiple laser pulses. All counters are synchronized with a synthesized $200\,\mathrm{MHz}$ clock signal which is also used to derive a $10\,\mathrm{MHz}$ clock signal for the target μP. We used an evaluation board [20] supporting a Xilinx Spartan-6 gate array [21]. Available for $35.

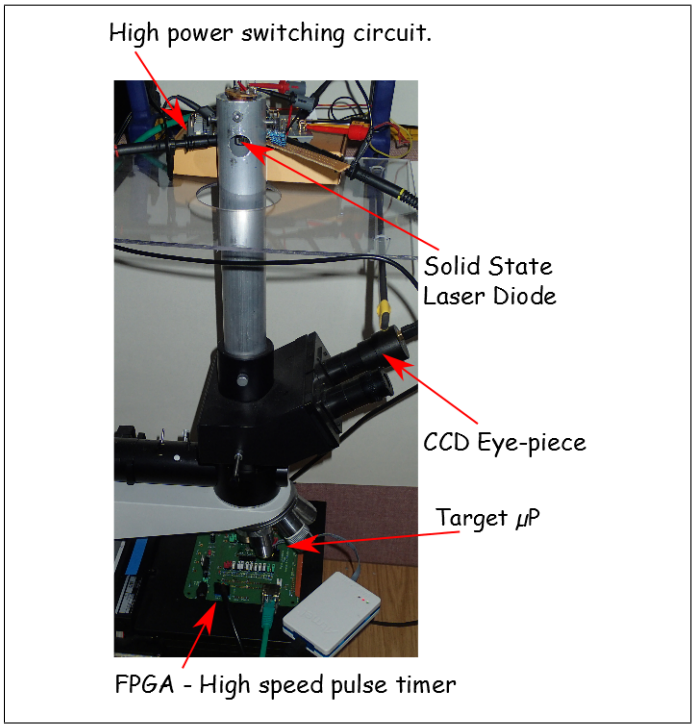

Fig. 1. Laser Station

The Laser diode was mounted vertically on the camera mounting point of a 40 year old trinocular Leitz SM-LUX HL microscope obtained via Ebay for approximately $250.

Equally consistent results were obtained with both the 10X or the 20X objective lenses and in the end we performed all of our characterization experiments using the Leitz Wetzlar NPL 10X lens.

One of the microscope's eye-pieces was replaced with a CCD camera to enable adjustments to be made while safely viewing the laser. This CCD camera also detects NIR light, suggesting the apparatus will also work for backside attacks. Focus and alignment of the laser was achieved by burning small holes in paper targets.

Total cost of the laser station, excluding the controlling PC and the board supporting the μP under investigation, was under $500.

B. Control

In this study we used an Atmel AVR-Tiny841 [22]. The AVR family of μPs is used in both smart cards and in SoCs targeting the IoT market. The μP was mounted on a PCB which itself mounted on the FPGA evaluation board. Both the μP and the FPGA can be reset and reprogrammed in-circuit by the controlling PC. The FPGA provides the 10 MHz clock for the μP and the μP provides a start signal that is used to synchronize the laser pulse generation counters with the executing test program.

For each experiment we loaded a specific program into the μP and repeatedly executed it with different pulse counts and pulse timings programmed into the FPGA. The output from each execution was fed back to the control PC via an RS232 interface for collection and analysis.

```
10:          breq _H
11: _L:      breq _LH
12: _LL:     breq _LLH
13: _LLL:    breq _LLLH
14: _LLLL:   nop
15:          nop
16:          nop
17:          nop
18:          ldi  r24,'0'
19:          ret
20: _LLLH:   nop
21:          nop
22:          nop
23:          ldi  r24,'1'
24:          ret
25: _LLH:    breq _LLHH
26: _LLHL:   nop
27:          nop
28:          nop
29:          ldi  r24,'2'
30:          ret
31: _LLHH:   nop
32:          nop
33:          ldi  r24,'3'
34:          ret
35: _LH:     breq _LHH
36: _LHL:    breq _LHLH
37: _LHLL:   nop
38:          nop
39:          nop
40:          ldi  r24,'4'
41:          ret
42: _LHLH:   nop
43:          nop
44:          ldi  r24,'5'
45:          ret
46: _LHH:    breq _LHHH
47: _LHHL:   nop
48:          nop
49:          ldi  r24,'6'
50:          ret
51: _LHHH:   nop
52:          ldi  r24,'7'
53:          ret
54:
55: _H:      breq _HH
56: _HL:     breq _HLH
57: _HLL:    breq _HLLH
58: _HLLL:   nop
59:          nop
60:          nop
61:          ldi  r24,'8'
62:          ret
63: _HLLH:   nop
64:          nop
65:          ldi  r24,'9'
66:          ret
67: _HLH:    breq _HLHH
68: _HLHL:   nop
69:          nop
70:          ldi  r24,'A'
71:          ret
72: _HLHH:   nop
73:          ldi  r24,'B'
74:          ret
75: _HH:     breq _HHH
76: _HHL:    breq _HHLH
77: _HHLL:   nop
78:          nop
79:          ldi  r24,'C'
80:          ret
81: _HHLH:   nop
82:          ldi  r24,'D'
83:          ret
84: _HHH:    breq _HHHH
85: _HHHL:   nop
86:          ldi  r24,'E'
87:          ret
88: _HHHH:   ldi  r24,'F'
89:          ret
```

Fig. 2. Branch Test Code

IV. Proof of Principle

This experiment had two aims. i) To test the capabilities of the equipment and ii) to confirm the predictions of earlier work, namely that multiple suitably timed laser pulses would induce multiple repeatable error effects.

We devised a simple branch matrix of four consecutive conditional branch instructions leading to sixteen different outcomes, see Figure 2.

In each test run the ZERO flag is set (or state **H**igh). The code would therefore be expected to take all four branches encountered, ultimately ending up at node HHHH, returning the value 'F'. Additional NOP operations in the code ensure that all paths take the same number of execution cycles to reach the end point. This enables us fire the laser at all relevant time intervals, as shown by the 'Pulse clk' signal in Figure 3, without hitting the code that reports the outcome. Figure 3 also shows the execution paths leading to the 16 possible outcomes.

By scanning all the possible timing patterns of $1 \ldots 4$ pulses we demonstrated that we could hit all 16 possible end states. As expected, where pulses coincide with the fetch cycle for a branch instruction we see branch skipping exactly as predicted. Where pulses coincide with the second cycle of a branch instruction or with one of the NOPs we see no effect. For example the '0' was reached consistently with the pulse timings of 3, 7, 11 & 15. We see five paths reach outcome '1', requiring 3 well timed errors. 1 trace of 3 pulses gets there as well as 4 traces of 4 pulses, where 3 of these pulses are well timed and the fourth has no effect.

TABLE I: Jump Matrix Termination States

Outcome	**Errors**[a]	**Samples**	**Paths**[b]
0	4	4	1
1	3	20	5
2	3	20	5
3	2	44	11
4	3	20	5
5	2	44	11
6	2	44	11
7	1	60	15
8	3	20	5
9	2	44	11
A	2	44	11
B	1	60	15
C	2	44	11
D	1	60	11
E	1	60	15
F	0	60	15
corrupt		0	0
Total		648	162

[a] Minimum number of errors required to reach this outcome. [b] Unique pulse patterns that yielded this outcome.

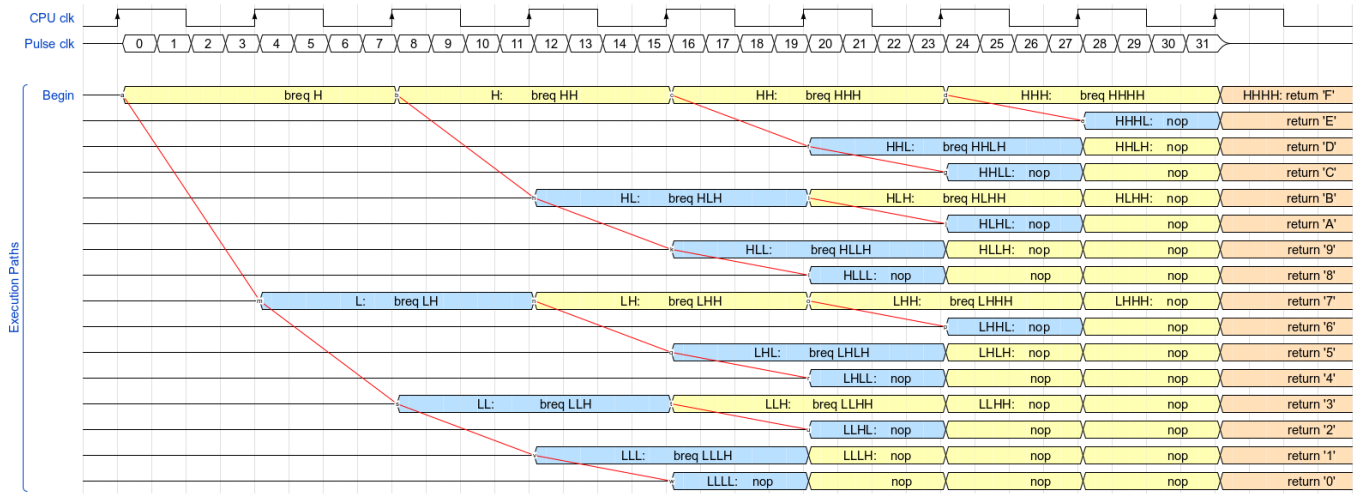

Fig. 3. Branch Test Execution Paths

From our earlier results [1] we had identified a specific physical location on the μP's upper surface where instruction skipping was reliably induced by a laser pulse synchronized with the third quarter cycle of the μP's clock. Table I shows the results for this location. Finding a 'sweet-spot' like this significantly reduces both the amount of data that needs to be captured and more significantly the time required to perform each experiment. The code under test took 8 cycles to execute and we injected all possible combinations of $1 \ldots 4$ pulses during this time frame. Each pulse pattern was repeated 4 times for good measure and this repetition proved unnecessary as each group of 4 samples consistently gave the same result.

These early results confirmed that the laser station was powerful enough and accurate enough to induce repeatable errors in the μP's execution path. As far as pulse timings and repetition rate are concerned it exceeded the capabilities of the equipment it was designed to replace, and did so for a fraction of the cost. Even though we chose to attack a single μP with visible light from the topside, results from [17] indicate that similar behaviour can be expected on differing architectures and also when attacked from the backside with NIR light as demostrated by [2].

V. Application

The μPs deployed in the consumer electronics world are usually readily available to developers. This means a would-be attacker has almost unlimited access to samples upon which to run test code and to find an appropriately sensitive region to focus the laser on. Locating such a 'sweet spot' is the first step in executing an attack.

Whilst the details of the mode of failure are interesting [15], such knowledge is not required when considering the consequences of an attack [2], and these attacks are therefore accessible to a wide range of attackers, in particular those with restricted resources.

It is unlikely that most attackers would have in-depth knowledge of the μP's internal layout, or understanding of the physical nature of the induced faults. However, as we show here, this is not necessary and a simple interpretation of the observed faults adequately categorizes the devices behaviour.

Working on this premise we devised a simplistic but highly relevant fault model. This enabled us to measure the efficacy of different software defensive structures without needing to understand the precise nature of the induced error.

A. Fault Model

We considered an executing program to be in one of four states as shown in Figure 4.

1) ***normal***: Execution as expected. Unaffected by error injection.
2) ***corrupted***: Execution continues within our program but some instructions may have been skipped and some values may be incorrect.
3) ***trapped***: The executing code has recognized it is in the *corrupted* state and deliberately entered a trap. Here it would be normal for a program to erase/protect valuable assets and freeze execution.
4) ***crashed***: The executing code is out of our program's control.

Transitions between the states occur as a consequence of one of three events.

1) ***crash***: may occur as a consequence of a single catastrophic error such as a skipped RET or as a result of continued execution with corrupt data for example RET from a function when the stack or frame pointers were previously corrupted.
2) ***skip***: occurs when an instruction fails to execute but the program continues. Here there is a high likelihood that some aspect of the program's state will be corrupt. It also possible that a fetched data value had been

corrupted as noted by [15]. Here we treat errors such as failure to execute an ADD or, adding the wrong value, as equivalent outcomes.

3) ***trap***: the executing program detects its own *corrupted* state. The efficacy and efficiency of this detection process is the primary driver behind this study.

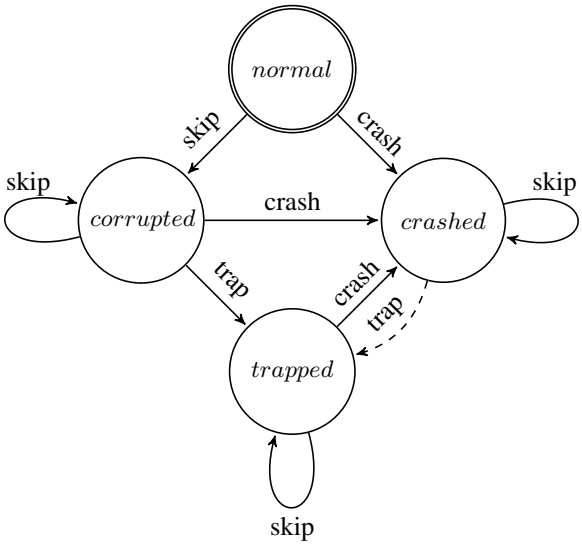

Fig. 4. Execution States.

Ideally programs will terminate in either the *normal* or the *trapped* states. The risks associated with termination in the *corrupted* state are well documented [3], [4]. Attempts to cause a crash during data delivery is a long established technique aimed at obtaining snapshots of a μP's memory contents. Clearly, termination in either *corrupted* or *crashed* states is undesirable.

Defensive coding therefore can be seen as a self-performed software sanity check aimed at detecting the *corrupted* state and entering the *trapped* state.

B. Evaluation of code defenses

We examined 15 code fragments, each employing a different defense strategy. The set of defenses, described below, was culled from an extensive portfolio of smart-card applications that have all been independently reviewed and evaluated for either EMV scheme or Common Criteria evaluation. Additional defenses were created following recommendations in [6] and [7] along with some described in [23] where a simulated attack was performed on code implementing a range of defenses.

The exhaustive testing of all pulse patterns is prohibitively time consuming thus some compromises have been made. The test time rises geometrically with the number of execution cycles to be examined. For a code fragment offering n intervals in which to inject a pulse, the number of samples to collect when using up to 4 pulses is given by

$$Samples = \sum_{p=1}^{4} \binom{n}{p} \qquad \text{where } \binom{n}{p} = \frac{n!}{p!(n-p)!}$$

Because of this the test code has been stripped down to the bare minimum. It has been observed that defensive code is frequently employed to protect small focused operations such as double testing within a comparison. Thus these short code samples still remain representative of real world defenses. It is also worth noting that many of the defenses become more effective when used in bigger more realistic modes. For example, A checksum over a small data block hardly differs from a duplicate data value, whereas when it is used over a larger block it is itself likely to be vulnerable to a miscalculation step. Thus signalling an attack even if the data it protects is uncorrupted.

Careful attention was paid to code output from the compiler. It was noted that, even with optimisations disabled, code was frequently in-lined and repeat code was often omitted. These compiler generated optimisations can totally remove defenses.

The code was also structured placing `SecretOp` code after the defended code that selectively accessed it. This arrangement was intended to expose the vulnerability of skipping the final RET operation and falling through into the protected code. It also engineered the scenario required for testing out-of-sequence execution detection.

The full set defenses examined is described below.

1) Unprotected: This sample acts as the reference behaviour for the other test samples. The basic logic is that a secret value is returned if a flag variable in RAM has a specific value. Under normal circumstances the secret value should not be returned. We would expect the secret value to be erroneously returned if either the pre-call test was corrupted or if the function return failed and execution was to 'fall through' into the `SecretOp` code.

```
// Unprotected

var Flag = FALSE

func test()

    ...
  if (Flag == TRUE)
    return SecretOp()
  else
    return EXPECTED
end

func SecretOp()

    ...
  return SECRET
end
```

Fig. 5. Unprotected

2) Double Test: Here the test is repeated. This defensive construct is frequently used in the EMV and JavaCard sample code that we reviewed. The technique is recommended in [6] with caveats. The rational is that if a test is skipped the repeat should detect the inconsistency and escape. The cost of this defense is trivial, it being one fetch and one conditional branch operation per decision point.

```
// Double Test

    ...
if (Flag == TRUE)
    if (Flag == TRUE)
        return SecretOp()
    else
        TRAPPED
else
    return EXPECTED
```

Fig. 6. Double Test

3) Retest in Target: This is a slightly more sophisticated variant of *Double Test*. Here the test is repeated within the called function. As above, it tests a property twice. Parameter related function call overheads will also add timing variability when this technique is used to protect multiple functions. This variation has the advantage of being able to detect out-of-order invocation of the `SecretOp` code. The cost of this defense is comparable to *Double Test*.

```
// Retest in Target

func SecretOp()
  if (Flag == TRUE)
      ...
      return SECRET
  else
      TRAPPED
end
```

Fig. 7. Retest in Target

4) Inverse Test: The test is repeated in its negative form. This has been recommended by evaluators as an improvement to the *Double Test* mechanism. The intention is that to reach the protected code a branch must be taken and another not taken. Thus attacks that rely on falling through conditional branches should fail. In practice the resulting assembler output does not reflect the 'C' source code's intention and we had to hand craft assembler code for this test. This makes the strategy impractical for large projects unless the compiler behaviour can be modified. The cost of this defense is equivalent to *Double Test*.

```
// Inverse Test

  ...
if (Flag == TRUE)
    if (Flag != TRUE)
        TRAPPED
    else
        return SecretOp()
else
    return EXPECTED
```

Fig. 8. Inverse Test

5) Double Data: This mechanism duplicates critical data variables in memory. The simple rational behind this mechanism is that if a variable becomes corrupted in memory or while being fetched it is unlikely that its shadow copy will be similarly corrupt. A variable's value is only trusted when both copies match. The runtime cost of this defense is equivalent to *Double Test* thus it has minimal impact of performance. The 100% duplication of data however is likely to prove impractical in resource constrained environments typical of μP deployments.

```
// Double Data

  ...
if ((FlagA == FALSE) &&
    (FlagB == FALSE))
  return EXPECTED
elseif ((FlagA == TRUE) &&
        (FlagB == TRUE))
  return SecretOp()
else
  TRAPPED
```

Fig. 9. Double Data

6) Data Inverse: This is a theoretical improvement on the *Double Data* mechanism. Here the shadow copy is the logical inverse of the primary data. The state variables are only trusted when the two copies are complementary. The presumption is that if both copies of a variable can be corrupted in memory or during a fetch then it is unlikely that the two corrupted instances will be mutually complimentary. The cost of this defense is the same as that for the *Double Data* defense.

```
// Inverse data

  ...
if (Flag != ~InvFlag)
    TRAPPED
elseif (Flag == TRUE)
    return SecretOp()
elseif (Flag == FALSE)
    return EXPECTED
else
  TRAPPED
```

Fig. 10. Data Inverse

7) Checksum: The 100% redundancy of the *Double Data*, and *Data Inverse* defenses can be avoided by maintaining a checksum over a set of variables. This checksum is verified before the variables are used and recalculated and set whenever the variables are updated. The runtime cost of this defense is very high as verifying and calculating checksums over blocks of data is computationally expensive. However for large data blocks it has a low demand on resources. This defense has been seen defending large blocks of critical data as a one time operation before a complex algorithm proceeds. For example to verify cryptographic keys.

```
// Checksum over data

  ...
CrcVerify_TrapOnError()

  ...
if (Flag == TRUE)
  return SecretOp()

else
  return  EXPECTED
```

Fig. 11. Checksum Data

8) Redundant Representation: This technique aims to detect corrupted data by inserting redundant data bits (sometimes referred to as sentinels) within a value's representation. If the sentential bits do not match the expected pattern then the value as a whole must have been corrupted. The technique can be used to encode multiple flags into a single word but is most frequently deployed to represent a single flag value where the sentinels and value can be tested in a single operation. The cost of this defense is very low. It requires very little storage as sentinels can be encoded within in the redundant bits of a variable's storage word. Similarly the testing of sentinel values can be performed in parallel with the associated data, or in the worst case, after logical masking and comparison operations.

```
// Redundant Representation

const STRUE  = 0xA5
const SFALSE = 0xA7

  ...
if (Flag == STRUE)
    return SecretOp()
elseif (Flag == SFALSE)
    return EXPECTED
else
    TRAPPED
```

Fig. 12. Redundant Representation

9) Repeat Calculation: Errors in computation can be detected by performing an operation twice and confirming that both calculations yield the same result. The technique is computationally inefficient but may be appropriate when invoking hardware assisted calculations using peripherals such as co-processors. Repetition has its own drawbacks and *Inverse Calculation* may be more appropriate.

```
// Repeated Calculation

  ...
u16 nTmp1 = SecretOp()
u16 nTmp2 = SecretOp()

if (nTmp1 != nTmp2)
    TRAPPED
else
   return nTmp2;
```

Fig. 13. Repeat Calculation

10) Modified Compensated: In this technique an input parameter affects the result of a calculation in a way that can be easily compensated for by the caller. This enables the caller to invoke a function multiple times, yielding different answers and still be able to confirm the accuracy of the results. If the function is entered accidentally during out-of-order processing then the returned value is likely to be modified by an unknown input. This provides an additional level of defense beyond the ability to check the accuracy of the calculation. The computational cost of this defense depends on the complexity of removing the input's bias from the result.

```
// Modified & Compensated

  ...
Tmp1 = Calculation(Rnd1)
Tmp2 = Calculation(Rnd2)
Tmp3 = Clear(Tmp1, Rnd1)
Tmp4 = Clear(Tmp2, Rnd2)

if (Tmp3 != Tmp4)
  TRAPPED
else
  return Tmp3
```

Fig. 14. Modified Compensated

11) Alternative Algorithm: This technique aims to overcome the primary weakness inherent in *Repeat Calculation*. Namely that the power profile of repeat calculations is likely to be similar and therefore recognizable. Thus enabling an attacker to synchronize attacks on the same moments in both invocations of a function. By using different algorithms it is less likely that an attacker could influence both to yield matching erroneous results. This is a very costly defense in terms of both code volume and processing time. Performing two calculations and comparing their results must take more than double the time of a single execution of the optimal algorithm.

```
// Alternative Algorithm

  ...
Tmp1 = Method1()
Tmp2 = Method2()

if (Tmp1 != Tmp)
    TRAPPED

else
  return Tmp2
```

Fig. 15. Alternative Algorithm

12) Inverse Calculation: For some algorithms the inverse calculation can be significantly quicker than the normal calculation. In this situation it is possible to confirm computational accuracy by ensuring the input data can be recovered and verified from the deliverable result. The confirmation step also avoids repetition of the primary computation. A surprising result is that the cost of this defense is not always as high as the *Alternative Algorithm* defense. For example with RSA signature generation; verifying the signature using the public key is significantly faster than repeating the signing calculation. Given the high cost of failure, see [3], this defense is frequently deployed.

```
// Inverse Calculation

  ...
Tmp1 = Method(Input)
Tmp2 = InvMethod(Tmp1)

if (Input != Tmp2)
  TRAPPED

else
 return Tmp1
```

Fig. 16. Inverse Calculation

13) Jump Id: This technique aims to detect out of order execution. Function entry and exit code is augmented with additional parameters to demonstrate the caller's intent to invoke the function. If the execution path accidentally enters the function, for example by skipping a RET and falling through to adjacent code then the executing function can recognize this is not deliberate and enter the *trapped* state. Similar behaviour at the function exit enables the caller to confirm the return occurred from the correct function. This defense is relatively expensive as each defended function's invocation, entry, exit & return state must be instrumented with various data set, get and comparison operations.

```
// Jump ID

func ProtectedFn()
  if (!IdVerify(CALL_F))
    TRAPPED
  else
    ...
    IdSet(RET_F)
    return Result
end

...
IdSet(CALL_F)
Val = ProtectedFn()
if (IdVerify(RET_F))
  TRAPPED

else
  return Val
```

Fig. 17. Jump Id

14) Waymark Late Test: Whenever execution passes a particular point a waymark variable is updated. At the end of the calculation the waymark can be examined to confirm that all the critical points of the proceeding execution path were executed. This technique is well suited for code containing loops and function calls where different fields within the waymark can be manipulated independently and where the final value is constant and predictable at compile time. The overhead is minimal and the test is performed at the end of the processing.

```
// Waymark - Late Test

WM = IV
  ...
WM += M1
  ...
WM += Mx
  ...
if (WM != IV+M1+...Mx)
  TRAPPED

else
   return nRetVal
```

Fig. 18. Waymark Late Test.

15) Waymark On the fly: An alternative waymark mechanism enables the early detection of unexpected or skipped code. Here, each time a waymark is updated, its current expected state is simultaneously verified. Frequent inline tests throughout the whole code body make this mechanism able to detect of out of order processing and it can be used to switch a *crashed* program to a *trapped* state. It is marginally slower than the *Waymark Late Test* variant but has the advantage of noticing the effects of skipped code at the earliest possible opportunity.

```
// Waymark - On the fly

func Waymark(n)
  if (n != nNextWM)
      TRAPPED
  else
    nNextWM++
end

...
Waymark(10)
...
Waymark(11)
...
Waymark(12)
return Result
```

Fig. 19. Waymark on the fly.

C. Test and Analysis

The test code was designed to return different values depending on the execution state at the end of the run. Three coded return values identified i) an expected value, ii) a secret value that should not be returned, and iii) an indicator to signal that a trap had been reached. These values corresponded to the *normal*, *corrupted* and *trapped* states of our execution state model. We we also treated unexpected return values as being corrupt. Failure to reply, or excessive data bursts, were treated as identifying the *crashed* state.

Each code sample was executed to determine the execution time of the algorithm under test and then subjected to all possible time and count combinations of $1 \ldots 4$ laser pulses within this time window. The number of samples we obtained per algorithm varied from the low thousands to hundreds of thousands as the execution times of the tests varied. The total number of samples collected and the corresponding terminating states are presented in Table II. These results are presented again in Fig 20 separating the results by the number of pulses injected and showing the percentage of the samples terminating in particular states.

The laser was focused on a previously identified 'sweet spot' that reliably caused the μP to misread memory fetches. This gave us a high probability that each laser pulse would cause some form of error, either instruction skip or a faulty operand fetch. Each pulse pattern was repeated several times during each run of the experiment.

TABLE II: Test Samples

Defense	C[a]	S[b]	Termination State N[c]	T[d]	Co[e]	Cr[f]
Undefended	11	2244	376	0	1420	448
Double test	12	3172	629	452	1451	640
Retest 'target	22	36432	9542	2363	15496	9031
Data inverse	15	7760	1212	2448	3072	1028
Checksum	33	187748	84212	30228	53138	20170
Inverse	11	2244	1019	0	605	620
Double data	18	16188	3608	5217	6694	669
Redundant	12	3172	1094	0	833	1245
Repeat calc.	32	165792	32967	70031	34576	28218
Mod. comp.	25	61100	11004	0	10262	39834
Alt. alg.	31	145824	31614	55853	26538	31819
Inv. calc.	46	717784	102436	153007	82206	380135
Jump id	30	127720	22430	59467	30495	15328
Waymark late	27	83412	19177	40321	16335	7579
Waymark 'fly	35	238140	32366	140350	54933	10491

[a] Instruction Cycles. [b] Samples. [c] Normal. [d] Trapped. [e] Corrupt. [f] Crashed.

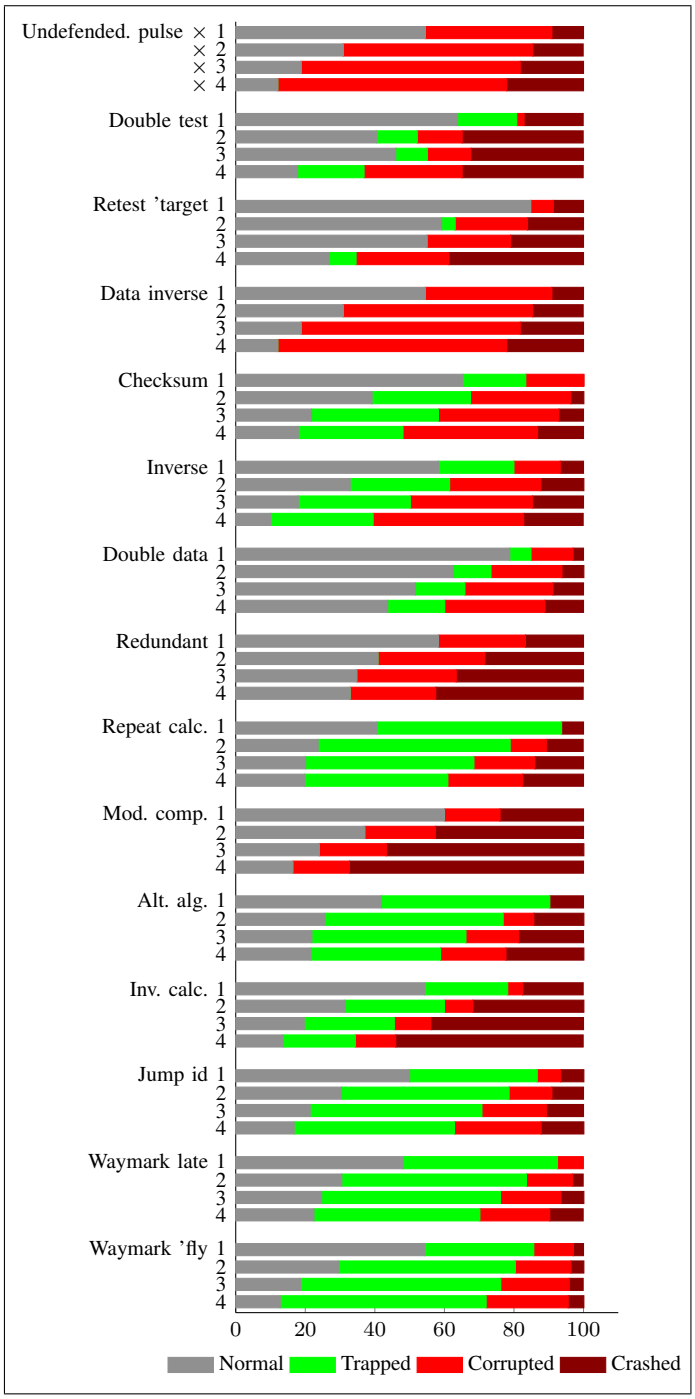

Fig. 20. Termination States

D. Results

Figure 20 shows the termination states of the test programs after $1 \ldots 4$ pulses were injected. For each run of the experiment the pulses were injected at all combinations of time intervals.

Two features immediately stand out and are worthy of further explanation.

– The large number of *normal* terminations initially appears to be surprising given the high probability of inducing errors at the chosen target site. Examination of the execution trace and the pulse times shows that these *normal* terminations occur when the pulses coincide with instructions that take multiple clock cycles to execute. Here the vulnerable pre-fetch is not performed on every cycle. Similarly on conditional branch operations the potentially erroneous pre-fetched fall-through option may be discarded in favour of the calculated branch-taken address.

– Secondly the trend for *crashed* to increase with the number of pulses. Similar analysis of the probable execution path and the pulse times suggests that execution entered the *trapped* state causing an early return from the test. In these cases the result reporting functionality is itself subjected to later pulses, resulting in erroneous result delivery and consequently being interpreted as *crashed*.

Clearly none of the defenses are infallible. By close examination of the pulse injection times and the presumed execution paths it is possible to infer modes of failure and strengthen the defenses accordingly. The most notable failure mechanisms seen across the range of tests are:

– Out-of-order processing. Here a section of code is executed at an unexpected time. For example 'fall-through', where, after skipping a RET operation, execution will continue into the neighboring function. In many cases the next RET encountered will mean the program returns to the original caller and resumes normal operation. Similar effects occur when a function call is skipped.
– Skipping of comparisons. This is a specific example of out-of-order processing when execution falls through to conditional code regardless of the state of the condition. All of the defenses ultimately rely on comparison operations to decide whether to proceed or not. This decision process is unrelated to the data representation or algorithm used. Disrupting this decision negates the defense and duplicated tests can be defeated by repeating the pulses.

We have seen that waymark based defenses offer a robust defense against out of order processing. In particular *Waymark on the fly*. The accumulative nature of the state representation means that even if one test is bypassed subsequent tests will still identify the error state. This property of waymarks also makes them effective at recognising and trapping a crashed program. Waymarks however cannot defend against erroneous calculations caused by the skipping of arithmetic or faulty storage access operations.

Based on these observations we combined the most effective defenses to create a testable hybrid defense. We used waymarks because they are computationally efficient and relatively effective at detecting skipped or out-of-order code; *Repeat Calculation* because it was quick while recognising an alternative data integrity test may be more appropriate depending on the algorithm being defended; and, the simple *Double Test* augmented with waymarks to verify the result. Finally the *trapped* state reporting code was also modified to ensure that the reporting of the termination state did not occur while pulses may still be expected.

```
// Hybrid Defense

func Calculation(WP)
    Waymark(WP)
    ...
    Waymark(WP+1)
    return EXPECTED

func TestEQ(WP, V1, V2)
    Waymark(WP)
    return (V1 == V2)

    ...
  A = Calculation(1)
  B = Calculation(3)
  if (TestEQ(5, A, B))
    if (TestEQ(6, B, A))
      Waymark(7)
      return A
  TRAPPED
```

Fig. 21. Hybrid Defense

The test program took 102 instruction cycles to execute resulting in $17,706,112$ samples that took 7 weeks to collect at 4 tests per second. Table III and Figure 22 show the results of this experiment.

TABLE III: Hybrid Test

Defense	C	S	Termination State N	T	Co	Cr
Hybrid	102	17706112	893790	16659489	2684	150149

Such comprehensive coverage of all pulse patterns was only possible after the discovery of the 'sweet spot' as it would be impractical to cover all timings over all physical locations. As a consequence we recognize we are looking at a specific single mode of failure. Earlier work on this particular μP suggests that the errors induced here relate to a corrupted read from non-volatile storage. This effect has also been noted by [16] on a different μP architecture. The phenomenon affects fetching of both instructions and data from non-volatile memory, but does not affect data fetches from RAM or Registers. Our test calculation deliberately employed non-volatile data fetching to mitigate the effect of this bias and ensure faulty data as well as code influenced the results.

We also rely on the time invariance to correlate pulse patterns with executing code. A jittering CPU clock would require the collection of many more samples but would ultimately be expected to give the same end state ratios.

VI. Conclusions

This study has demonstrated that intricate error injection attacks combining multiple pulses and high repetition rates with precision timing can be achieved using low cost and readily available components. As a tool for error injection attacks our budget device's capabilities far exceed those of the expensive YAG laser cutter it replaces. Thankfully this same equipment can be used to characterize the efficacy of software defenses. This characterization can then be used to prescribe efficient combinations of defenses, thereby removing much of the guesswork currently involved in formulating defensive code.

This study has highlighted the generic flaw in most defenses relating to data accuracy. Namely that they boil down to a final decision point. Redundant data representations and repeated/modified calculations, ultimately rely on a late go/no-go decision. This is their Achilles' heel. Here we have

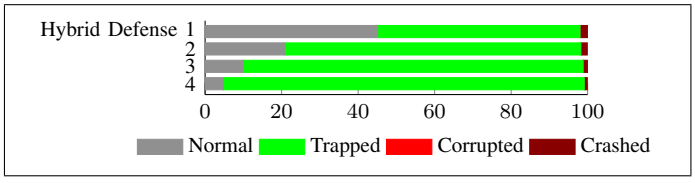

Fig. 22. Hybrid defense termination states

demonstrated an effective double test construct that exploits the strengths of waymarking to provide additional confidence that these tests are being performed.

This study has also emphasized the importance of freezing execution as soon as erroneous behaviour has been detected. Continuing execution within a *trapped* state gives the attacker an extended opportunity to induce a compromising error.

No software defense can be infallible in an environment where each and every μP operation is effectively optional. In common with cryptography, the strength of software defenses ultimately lies in the practical infeasibility of testing all combinations.

The low cost and relative ease of construction of our laser error injector suggests that developers of IoT devices need to seriously consider the likelihood and consequences of an attack on their products. This study should encourage IoT developers to use defensive coding as normal practice, and in many cases consider using devices with physical defenses against this category of attack. This is crucial, because it must be assumed that these attack techniques are readily available to criminals, malicious attackers, and amateur hackers.

REFERENCES

[1] M. S. Kelly, K. Mayes, and J. F. Walker, "Characterising a cpu fault attack model via run-time data analysis," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 79–84, IEEE, 2017.

[2] F. Schellenberg, M. Finkeldey, N. Gerhardt, M. Hofmann, A. Moradi, and C. Paar, "Large laser spots and fault sensitivity analysis," *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 203–208, 2016.

[3] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults (extended abstract)," in *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding* (W. Fumy, ed.), vol. 1233 of *Lecture Notes in Computer Science*, pp. 37–51, Springer, 1997.

[4] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings* (B. S. K. Jr., ed.), vol. 1294 of *Lecture Notes in Computer Science*, pp. 513–525, Springer, 1997.

[5] EMVCo, LLc, *Issuer and Application Security Guidelines*, November 2007.

[6] MasterCard Worldwide, *Security Guidelines for M/Chip Advance Developers*, 5 March 2010.

[7] Visa Inc., *Visa Security Guidelines - Multi-application Platforms*, March 2009.

[8] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on rsa with crt: Concrete results and practical countermeasures," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 260–275, Springer, 2002.

[9] D. H. Habing, "The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits," *IEEE Transactions on Nuclear Science*, vol. 12, no. 5, pp. 91–100, 1965.

[10] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers* (B. S. K. Jr., Ç. K. Koç, and C. Paar, eds.), vol. 2523 of *Lecture Notes in Computer Science*, pp. 2–12, Springer, 2002.

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

[12] P. Maurine, K. Tobich, T. Ordas, and P. Y. Liardet, "Yet Another Fault Injection Technique : by Forward Body Biasing Injection," in *YACC'2012: Yet Another Conference on Cryptography*, (Porquerolles Island, France), Sept. 2012.

[13] M. Tunstall, "Attacks on smart cards." Web, http://www.wisdom.weizmann,ac.il/-tromer/acoustic, 2008.

[14] N. Vashistha, M. T. Rahman, O. P. Paradis, and N. Asadizanjani, "Is backside the new backdoor in modern socs?: Invited paper," in *2019 IEEE International Test Conference (ITC)*, pp. 1–10, Nov 2019.

[15] J. Proy, K. Heydemann, F. Majéric, A. Cohen, and A. Berzati, "Studying em pulse effects on superscalar microarchitectures at isa level," *arXiv preprint arXiv:1903.02623*, 2019.

[16] B. Colombier, A. Menu, J.-M. Dutertre, P.-A. Moëllic, J.-B. Rigaud, and J.-L. Danger, "Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller?," *IACR Cryptology ePrint Archive, Report 2018/1042 (2018)*, 2018.

[17] O. M. Guillen, M. Gruber, and F. De Santis, "Low-cost setup for localized semi-invasive optical fault injection attacks," in *Constructive Side-Channel Analysis and Secure Design* (S. Guilley, ed.), (Cham), pp. 207–222, Springer International Publishing, 2017.

[18] Nichia Corporation, Tokushima 77-8601, Japan, *Datasheet - Specifications for Nichia BULE laser diode bank.* NUBM08, UTZ-SF0119E.

[19] iC Haus Gmbh., *Datasheet - iC-HG. 3A Laser Switch*, 2014. Rev. B2.

[20] Numato Systems, LLC, *Datasheet - Spartan-6 Family Overview.*, February 2016. Mimas - Spartan 6 FPGA Development Board, https://numato.com/docs/mimas-spartan-6-fpga-development-board/.

[21] Xilinx Inc, *Datasheet - Spartan-6 Family Overview.*, October 2011. DS160 (v2.0).

[22] Atmel Corporation, *ATtiny841 Datasheet – 8-bit AVR Microcontroller with 4/8K Bytes In-System Programmable Flash*, 05 2014. Rev. 8495H.

[23] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013* (E. Macii, ed.), pp. 404–409, EDA Consortium San Jose, CA, USA / ACM DL, 2013.