# 

Fault Mitigation Patterns

By Marc Witteman, [witteman@riscure.com](mailto:witteman@riscure.com)

Joshua Beaker

Table of Contents

[1. Introduction 4](#_Toc71664014)

[2. Protect 4](#_Toc71664015)

[Pattern 1. FAULT.MINIMIZE.BRANCHES 5](#_Toc71664016)

[Pattern 2. FAULT.RANDOM.DELAY 6](#_Toc71664017)

[3. Respond 7](#_Toc71664019)

[Pattern 3. FAULT.DETECT 7](#_Toc71664020)

[Pattern 4. FAULT.PENALTY 8](#_Toc71664022)

[4. Recover 9](#_Toc71664024)

[Pattern 5. FAULT.FLOW.CONTROL 9](#_Toc71664025)

[Pattern 6. FAULT.DECISION.CHECK 12](#_Toc71664028)

[Pattern 7. FAULT.BRANCH.CHECK 13](#_Toc71664030)

[Pattern 8. FAULT.VALUE.CHECK 14](#_Toc71664032)

[Pattern 9. FAULT.AGGREGATE.CHECK 15](#_Toc71664033)

[Pattern 10. FAULT.LOOP.CHECK 15](#_Toc71664035)

[Pattern 11. FAULT.CRYPTO.CHECK 16](#_Toc71664037)

[5. Conclusion 17](#_Toc71664038)

[6. Reference 17](#_Toc71664039)

**Abstract**: Hardware Fault Attacks can break software security by revealing secrets during program execution, or changing the behaviour of a program. Without profound knowledge of these attacks, it is hard to defend code effectively. Fault resistance requires pervasive protection throughout the code. This paper introduces a collection of secure programming patterns for security critical devices. These patterns help developers to mitigate the risk of fault injection in a cost effective way.

# Introduction

Hardware Fault Injection is a class of hardware security attacks which have become increasingly popular, as these attacks are powerful and have a high probability of success. Most devices today are completely vulnerable against these attacks as developers have little awareness of the threat, and do not know how to protect their code. Software security implications of these attacks are discussed in [1].

Fortunately, it is possible to harden software and mitigate the Fault Injection threat. However, since the number of attacker opportunities is very high this could require a large effort. In this paper, we propose a set of 11 fault mitigation patterns. These patterns have the advantage that they can be repeatedly applied, without making a detailed design for each instance, and thus minimize the mitigation effort.

We organize the patterns along five main strategies:

1. **Identify**: tests to increase visibility in hardware and software to find possible faults;
2. **Protect**: increase code resistance such that fault are less likely to disturb program behavior;
3. **Detect**: measures to alert your system to the presence of a fault;
4. **Respond**: actions to deter attackers after detecting a fault;
5. **Recover**: code resilience to prevent insecure behavior following a fault;

The remainder of this paper details the patterns which implement the protect, respond, and recover strategies. We include illustrative examples in the C language (translation to other languages should not be hard). These examples serve to clarify the pattern, and can be adapted and applied for instances of the problem.

It is our experience that these patterns, if well understood, are applied efficiently at the average cost of 12 minutes per instance. Furthermore, there is typically no need to protect the entire code base: protecting the critical code is often good enough.

We test the effectiveness of each pattern against two fault models, namely bit flips in instructions and instruction skipping. These tests are executed on code compiled for the ARMv7 architecture.

# Protect

An attacker using fault injection aims for a specific effect, e.g. changing a value or decision. Protection is introduced by making useful values hard to achieve, or by making it hard to find the right timing to manipulate a decision.

## FAULT.MINIMIZE.BRANCHES

|  |  |
| --- | --- |
| Problem | Branching provides an opportunity for attackers to skip the branch, sending execution past entire functions and checks. Addresses can also be flipped at branches to send execution into “random” parts of the code, sometimes to the benefit of an attacker. |
| Solution | By using inline compiler flags we can limit the amount of branches due to functions drastically. |
| Reduction | Complete reduction in branch flaws from missing function calls. 75% branch related successful fault reduction in our tests. |

The following is an example of a bad implementation that leads to vulnerabilities:

*//bad implementation*

void main(…){

method1();

method2();

*//continue critical code execution*

}

void method1(…){

*//critical code execution 1*

}

void method2(…){

*//critical code execution 2*

}

The following is an implementation of the countermeasure that reduces vulnerabilities:

**Example Minimize Branches**

*//good implementation*

*//bad implementation*

void main(…){

method1();

method2();

*//continue critical code execution*

}

inline void \_\_attribute\_\_ ((always\_inline)) method1(…){

*//critical code execution 1*

}

inline void \_\_attribute\_\_ ((always\_inline)) method2(…){

*//critical code execution 2*

}

## FAULT.RANDOM.DELAY

|  |  |
| --- | --- |
| Problem | Attackers aim to manipulate a specific value or decision. |
| Solution | Insert random-length delays throughout code making it much harder to hit a specific moment in software execution. |
| Effectiveness | N/A |

## The following is an implementation of the countermeasure that reduces vulnerabilities:

**Example Random Delay**

void delay () { *// wait random time*

int loops = rand() & 0x3FF; *// 10 bit entropy, 1024 possible values*

while (--loops >= 0 ) {

loops++;

loops--; *// loop counter changes avoid compiler removal of non-functional loop*

}

}

# Respond

Immunity is hard. Especially when so many vulnerabilities exist. It is therefore important to detect fault attempts, and act accordingly.

## FAULT.DETECT

|  |  |
| --- | --- |
| Problem | Repeated fault attempts may eventually be successful. |
| Solution | Test occurrence of logically impossible behaviour to discover fault attempts. Include randomly occurring traps, deliberate verifications or known values, to complicate avoidance of detection. |
| Reduction | N/A |

## The following is an implementation of the countermeasure that reduces vulnerabilities:

**Example Fault Detect**

#define SUCCESS = 0x3CA5C35A

#define INV\_ SUCCESS 0xC35A3CA5

if (conditionalValue == SUCCESS) { *// enter critical path*

trap(); *// enter random delay where attack can be detected*

if (~conditionalValue == INV\_ SUCCESS) { *// check complement*

*// continue* critical code *execution*

} else fault\_detected(); *// fault detected*

}

void trap () { *// wait random time and catch faults*

int loops = rand() & 0x3FF; *// 10 bit entropy, 1024 possible values*

while (--loops >= 0 ) {

if (~SUCCESS != INV\_ SUCCESS) fault\_detect(); *// fault detected!*

}

}

void fault\_detected() { *// what to do when a fault is detected*

exit(); *// terminate program*

}

## FAULT.PENALTY

|  |  |
| --- | --- |
| Problem | Immunity against fault injection is hard, especially given the widespread nature of the problem. Attackers will therefore be inclined to repeat fault experiments and keep trying to find an exploitable weakness. |
| Solution | By introduction of a penalty, it is possible to deter attackers and reduce their chance of success. Such penalty can be in slowing down the system, or disabling functionality. The disabling functionality can be temporary (e.g. require service centre intervention), or permanent (termination or key zeroing). |
| Reduction | N/A |

## The following is an implementation of the countermeasure that reduces vulnerabilities:

**Example Fault Penalty**

#define SUCCESS = 0x3CA5C35A

#define INV\_ SUCCESS 0xC35A3CA5

int fault\_penalty\_delay = 0;

if (conditionalValue == SUCCESS) { *// enter critical path*

penalty\_delay(); *// include delay if attack detected earlier*

if (~conditionalValue == INV\_ SUCCESS) { *// check complement*

*// continue* critical code *execution*

} else fault\_detected(); *// fault detected*

}

void penalty\_delay () { *// wait random time and catch faults*

if (fault\_penalty\_delay <= 0) return; *// no attack penalty needed*

fault\_penalty\_delay--; *// decrement penalty waits*

sleep(10000); *// sleep 10 seconds*

}

void fault\_detected() { *// what to do when a fault is detected*

fault\_penalty\_delay = 10; *// introduce long waiting time in next 10 sessions*

exit(); *// terminate program*

}

# Recover

Even when fault injection is successful, it is possible to make resilient code that would continue correct execution and prevent exploitation. This includes double-checking to verify value correctness and crypto results, but also double-checking conditional statements, branches, loops, and program flow.

## FAULT.FLOW.CONTROL

|  |  |
| --- | --- |
| Problem | Fault injection can hit the program counter or stack. This can result in “code hijacking”, i.e. unauthorized jumps to privileged code. Verification of the correct flow should be done during the execution of sensitive code. |
| Solution | Use a counter to keep track of the correctness of the execution path. Check counter to verify completion of execution path. Prevent separate paths reaching identical counter values by stepping with prime numbers. |
| Reduction | Complete recovery of faults targeting method functions. |

The following is an example of a bad implementation that leads to vulnerabilities:

#define METHOD1\_STEP 13

#define METHOD2\_STEP 17

int counter;

void main( int argc, char\*\* argv ) {

counter = 0;

*method1( ... );* // call nested method

*// bad implementation*

// continue critical code execution

}

}

void method1( ... ) {

counter += METHOD1\_STEP; *// increase counter*

method2( ... );

counter += METHOD1\_STEP; *// increase counter*

}

void method2( ... ) {

counter += METHOD2\_STEP; *// increase counter*

// do other stuff

}

## The following is an implementation of the countermeasure that reduces vulnerabilities:

**Example Flow Control**

#define METHOD1\_STEP 13

#define METHOD2\_STEP 17

int counter;

void main( int argc, char\*\* argv ) {

counter = 0;

*method1( ... );* // call nested method

*// good implementation*

if (counter == 2\*METHOD1\_STEP + METHOD2\_STEP) { // check flow counter

// continue critical code execution, knowing the full path was taken

}

}

void method1( ... ) {

counter += METHOD1\_STEP; *// increase counter*

method2( ... );

counter += METHOD1\_STEP; *// increase counter*

}

void method2( ... ) {

counter += METHOD2\_STEP; *// increase counter*

// do other stuff

}

## 

## FAULT.DECISION.CHECK

|  |  |
| --- | --- |
| Problem | Sensitive data is manipulated by fault injection attacks. When a decision is made upon a single test the decision may be corrupted. |
| Solution | Double-check sensitive conditions. A conditional process based on sensitive data should double check the data. Preferably, these checks should not be identical, but complementary, as the attacker will have to perform two different types of attack. In addition, the checks should be temporally be separated to reduce the risk of a single multi cycle glitch bypasses check and double-check. Use a volatile complement to avoid compiler optimizations. Use the volatile keyword to avoid compiler optimizations |
| Reduction | 89% decrease in successful decision related faults |

The following is an example of a bad implementation that leads to vulnerabilities:

if (condition == SUCCESS) { *// enter critical path*

*// bad implementation*

*// continue* critical code *execution*

}

## The following is an implementation of the countermeasure that reduces vulnerabilities:

**Example Decision Check**

if (condition == SUCCESS) { *// enter critical path*

*// good implementation*

int volatile complement = ~SUCCESS;

if (~condition == complement) { *// check complement*

*// continue* critical code *execution*

}

}

## FAULT.BRANCH.CHECK

|  |  |
| --- | --- |
| Problem | Fault injection attacks manipulate branch choices. |
| Solution | Verify each branch. Use a different value for verification. Provide a failure value instead of a default value to avoid branch faults and ensure valid behaviour. Use volatile values for complements to stop the compiler optimizing out comparisons. |
| Reduction | 86% decrease in successful branch related faults |

The following is an example of a bad implementation that leads to vulnerabilities:

int volatile inv\_state = ~state; *// create bitwise complement for verification*

*// bad implementation*

switch (state) {

case INIT:

// continue critical code execution

case OPS:

// continue critical code execution

default:

//try to fail

}

## The following is an implementation of the countermeasure that reduces vulnerabilities:

**Example Branch Check**

int volatile inv\_state = ~state; *// create bitwise complement for verification*

*// good implementation*

switch (state) {

case INIT:

if (inv\_state == ~INIT) { *// verify inv\_state*

*// continue* critical code *execution*

} break;

case OPS:

if (inv\_state == ~OPS) { *// verify inv\_state*

*// continue* critical code *execution*

} break;

case FAIL:

*// fail intentionally*

break;

}

## FAULT.VALUE.CHECK

|  |  |
| --- | --- |
| Problem | Sensitive data is manipulated by a fault injection attack at any time during program execution. |
| Solution | Verify sensitive data. Sensitive data can for instance be protected by a checksum. Data protected in this way should be verified at regular intervals. Ideally, the integrity of sensitive data should be verified each time when used. Use the volatile keyword to avoid compiler optimizations. |
| Effectiveness | 70% reduction in successful faults related to result values. |

**Example Value Check**

int result = SOME\_VALUE; *// sensitive value assigned*

int volatile checksum = ~ SOME\_VALUE; *// use complement as checksum*

if (checksum == ~result) { *// verify checksum*

// continue critical code execution

}

## FAULT.AGGREGATE.CHECK

|  |  |
| --- | --- |
| Problem | Aggregated checks may be skipped by a fault, and render all included checks useless. |
| Solution | Do not aggregate multiple checks into one. |
| Reduction | 50% reduction in successful faults related to double checks. |

The following is an example of a bad implementation that leads to vulnerabilities:

*// bad implementation*

int aggregated\_result = (check() == TRUE && check() == TRUE);

if (aggregated\_result) { // single glitch may corrupt this statement

// continue critical code execution

}

## The following is an implementation of the countermeasure that reduces vulnerabilities:

**Example Aggregated Check**

*// good implementation*

if (check() == TRUE && check() == TRUE) { // safe double check

// continue critical code execution

}

## FAULT.LOOP.CHECK

|  |  |
| --- | --- |
| Problem | Repetitive processes running in a loop are terminated early by a fault injection attack. |
| Solution | Verify loop completion, avoid early or late completion. |
| Reduction | 48% reduction in successful loop based faults. |

The following is an example of a bad implementation that leads to vulnerabilities:

int i;

for ( i = 0; i < n; i++ ) { *// important loop that must be completed*

. . .

}

*// bad implementation*

// continue critical code execution

## The following is an implementation of the countermeasure that reduces vulnerabilities:

**Example Loop Check**

int i;

for ( i = 0; i < n; i++ ) { *// important loop that must be completed*

. . .

}

*// good implementation*

if (i == n) { *// loop completed*

// continuecritical code *execution*

}

## FAULT.CRYPTO.CHECK

|  |  |
| --- | --- |
| Problem | Cryptographic algorithms are sensitive to fault injection. They may even reveal key data through the output of false encryptions due to fault injection. Differential fault analysis (DFA) is a technique for this. |
| Solution | Check for fault injection during or after crypto. Verify any ciphered data before transmission by deciphering or repeated enciphering. If the deciphered data matches the original input, or the repeated enciphering matches the original output, it is most likely that the encryption was not corrupted. |
| Reduction | 39% reduction in successful flow related faults dependent on encryption. |

**Example Crypto Check**

encrypt( source, destination ); // result is in destination

decrypt( destination, source\_copy ); *// source\_copy should be same as source*

if (memcmp( source, source\_copy, length) == 0) { *// verify decryption*

send( destination ); *// continue* critical code *execution*

}

# Conclusion

Fault Injection is a growing threat to devices that need to be secure in the field. Since the amount of FI vulnerabilities in software is overwhelming, there is a need for a systematic mitigation approach. We propose to use Fault Mitigation Patters, a set of 11 countermeasures that can be applied throughout the code, and require little adaption for repeated application.

# Reference

[1] Bilgiday Yuce, Patrick Schaumont, Marc Witteman, *Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation*, <https://arxiv.org/pdf/2003.10513.pdf>



Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure BV

Frontier Building, Delftechpark 49

2628 XJ Delft

The Netherlands

Phone: + 31 (0) 15 251 4090

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure North America

550 Kearny St. Suite 330

San Francisco, CA 94108

USA

Phone: +1 (650) 646 9979

[inforequest@riscure.com](mailto:inforequest@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)

Riscure China

Room 2030-31, No. 989, Changle Road

Shanghai 200031

China

Phone: +86 21 5117 5435

[inforcn@riscure.com](mailto:inforcn@riscure.com)