Verifying Redundant-Check Based Countermeasures: A Case Study

Thibault Martin
Université Paris-Saclay, CEA, List
Palaiseau, France
thibault.martin@cea.fr

Nikolai Kosmatov
Thales Research and Technology
Palaiseau, France
nikolaikosmatov@gmail.com

Virgile Prevosto
Université Paris-Saclay, CEA, List
Palaiseau, France
virgile.prevosto@cea.fr

ABSTRACT

To thwart fault injection based attacks on critical embedded systems, designers of sensitive software use redundancy based countermeasure schemes. In some of these schemes, critical checks (i.e. conditionals) in the code are duplicated to ensure that an attacker cannot bypass such a check by flipping its result in order to get to a protected point (corresponding e.g. to a successful authentication or code integrity verification). This short paper presents a source-code-level verification technique of the correct implementation of such countermeasures. It is based on code instrumentation and deductive verification. The proposed technique was implemented in a tool prototype and evaluated on a real-life case study: the bootloader module of a secure USB storage device called WooKey, supposed to be resistant to fault injection attacks. We were able to prove the correctness of almost all redundant-check countermeasures in the module except two, and found an error in one of the unproven ones.

CCS CONCEPTS

• Security and privacy → Logic and verification; **• Software and its engineering → Formal software verification**;

KEYWORDS

Fault injection attacks, software countermeasures, deductive verification, Frama-C verification platform.

ACM Reference Format:
Thibault Martin, Nikolai Kosmatov, and Virgile Prevosto. 2022. Verifying Redundant-Check Based Countermeasures:, A Case Study. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event,* . ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3477314.3507341

1 INTRODUCTION

Context. Physical attacks of critical embedded systems (via light pulses, laser shots, clock, voltage or electromagnetic glitches, etc.) consist in causing a fault that alters correct execution of software [4, 6]. A frequent goal of such attacks is to bypass some critical checks in the code (such as user authentication, software integrity or software authentication checks) in order to get to a protected point that gives access to sensitive information or physical resources.

To counter such attacks, designers of embedded software use in particular redundancy based countermeasure schemes [4, 5]. In some of these schemes, critical checks (i.e. conditional statements, or tests) in the code are duplicated. In this way, if attackers manage to bypass one check by injecting a fault and flipping the result of the check, the redundant check still prevents from reaching the protected point. This countermeasure assumes that it is unlikely to inject two faults by physical attacks during the same execution in a coordinated way. It can be generalized to any number $k \geq 1$ of coordinated faults: if an attacker is assumed to be able to introduce k coordinated faults, each critical check should be repeated $k + 1$ times. For simplicity, in the examples of the paper we use $k = 1$.

Note that, from a strict C standard point of view, these countermeasures are dead code. Hence, as stated for instance in the description of the Common Software Weakness CWE-733[1] and in the last release of the C coding rules [2] by the National Cybersecurity Agency of France (ANSSI), developers should ensure that optimisations enabled in the compilation toolchain do not eliminate such manually added software countermeasures. This point is beyond the scope of this paper.

Examples. A simple C code with a redundant-check countermeasure is illustrated by Fig. 1. Assuming `password` is a user-submitted password and `secret` is the correct password, the duplicated conditional ensures that a bad password will be detected even if one of the conditions is inverted by an attack. Figure 2 shows a more interesting example, with redundant code integrity checks. Such a check is performed by function `check_code_integrity`. As a protection to bit flipping, this function returns a value of the `secbool` type, whose values `sectrue` and `secfalse` have a maximal bit-distance. The second condition is written in a different way, and is erroneous here: the developer should have used a bitwise negation `~chck2` instead of a logical negation `!chck2`. If `chck2` is `secfalse`, its logical negation is in fact 0 so that the test on line 8 is always false. Hence, if an attacker manages to flip the result of only the test on line 6, they will execute the protected line 9 even if code integrity check fails. This example illustrates an incorrect countermeasure, due to a misuse of `secbool` values. Other cases of wrong countermeasures are described below.

Motivation. Due to their redundant behavior, a correct implementation of countermeasures is difficult to verify, yet crucial to ensure resistance to the considered faults. Various approaches are used to assess the efficiency of countermeasures on a given system. Fault injection based techniques—reproducing potential physical attacks on the target device—allow validation engineers to detect

SAC '22, April 25–29, 2022, Virtual Event,

ACM ISBN 978-1-4503-8713-2/22/04.
https://doi.org/10.1145/3477314.3507341

---

[1] https://cwe.mitre.org/data/definitions/733.html

```c
if(password != secret) return 1; // Error, bad password
if(password != secret) return 1; // Error, bad password
// Protected: Successfully authenticated
```

Figure 1: Password check with a countermeasure.

```c
typedef enum {secfalse = 0x55aa55aa,
  sectrue = 0xaa55aa55} secbool; // secure true/false values
secbool check_code_integrity(); //checks code integrity
int main(){
  secbool chk1=check_code_integrity();
  if(chk1 != sectrue) return 1; // Error, compromised code
  secbool chk2=check_code_integrity();
  if(!chk2 == sectrue) return 1; //incorrect countermeasure
  // Protected: Successful code integrity check
}
```

Figure 2: Integrity check with a countermeasure.

(confirmed) vulnerabilities or get confidence that the system is sufficiently resistant to attacks. Such techniques have the advantage to consider the real-life target system, but remain costly and time-consuming, and cannot guarantee that the system will resist to similar attacks in a slightly different setting (e.g. different signal force, frequency, duration or number of attempts). Another approach consists in searching potential attacks at software level, by simulating a chosen set of possible faults in the code and trying to identify potential attacks using test generation [12] or its combination with static analysis [10], or to prove their absence using formal verification [7, 8]. Even if their results are subject to assumptions (about the considered fault model, fault simulation approach, compiler, etc.), software-level approaches provide a useful complement to physical evaluation: they are cheaper, can be fully automatic and can rigorously consider all potential faults with respect to the chosen fault simulation. Such techniques help to find hybrid software/hardware attacks [1].

This study continues previous efforts [7, 8, 10, 12] in this direction. We consider a simple fault model that allows the attacker to invert any subset of at most k checks in the code. "Test inversion" is seen as a very useful mode of fault simulation in a recent joint report by the French certification and evaluation authorities [1, Sec. 16.4].

Contributions. This short paper presents a source-code-level formal verification technique of correct implementation of redundant-check based countermeasures. Its purpose is to prove that provoking up to k test inversions in the code should not allow an attacker to reach the protected code. It includes two steps: a dedicated code instrumentation simulating possible faults in critical checks ("test inversions") by mutations; and deductive verification of the resulting code trying to formally prove that the countermeasures effectively prevent attacks. The proposed technique was implemented inside LTest[2] [3], an open-source testing toolset, and relies on the Frama-C[3] verification platform [9]. We evaluated this technique on a real-life case study: the bootloader module of a secure USB storage device called WooKey[4], implemented by the ANSSI and supposed to be resistant to fault injection attacks. We were able to formally prove the correctness of all redundant-check countermeasures in the module except two, and found an error in one of the remaining ones. This error remained undetected despite the fact that this module was rigorously analyzed by 10 evaluation centers[5] as part of a recent evaluation challenge [1]. It confirms the interest of the proposed dedicated approach.

```c
if(C_1)
  {code1;}
...
if(C_N)
  {codeN;}

//Protected
```

→

```c
int mut_1 = mutated();
if((!mut_1 && C_1) || (mut_1 && !C_1))
  {code1;}
...
int mut_N = mutated();
if((!mut_N && C_N) || (mut_N && !C_N))
  {codeN;}
/*@ check !mut_1 && ... && !mut_N;*/
//Protected
```

Figure 3: (a) A code example, and (b) its automatic annotation by mutations for fault simulation.

```c
#define MAX_MUTATION 1 // Max number of modeled faults
unsigned int cpt_mut = 0;
/*@
  assigns cpt_mut;
  behavior cannot_mutate:
    assumes cpt_mut ≥ MAX_MUTATION;
    ensures !\result;
    ensures cpt_mut == \at(cpt_mut, Pre);
  behavior can_mutate:
    assumes cpt_mut < MAX_MUTATION;
    ensures  \result ⇔ cpt_mut == \at(cpt_mut,Pre) + 1;
    ensures !\result ⇔ cpt_mut == \at(cpt_mut,Pre);
*/
int mutated();
```

Figure 4: Uninterpreted function `mutated` and its contract.

2 VERIFICATION APPROACH

Overview. Our verification approach proceeds as follows. The user indicates the beginning and the end of the critical sections of the code, inside which all tests should resist to fault injection attacks. This is done to focus only on the critical steps (containing authentication, integrity checks, version control, etc.) since other parts of the code (e.g. following the authentication) are typically less critical and do not integrate countermeasures. The end of the indicated code segment corresponds to the protected point that gives access e.g. to sensitive resources or information. In a critical code segment, all checks are instrumented to simulate faults introduced by an attacker according to the considered fault model: an attacker can invert up to k tests. The target property is expressed as an annotation in the Acsl[6] specification language and states that no attack can reach the protected point. Then, a deductive verification tool is run on the instrumented code to try to formally verify this annotation. Figure 3b presents the instrumentation scheme for a critical part of code, given in Fig. 3a, with N tests. We will now detail its main components.

Fault Simulation. To simulate a possible fault injection, we introduce a specific function, called `mutated`, for which we provide only

---

[2] https://github.com/ltest-dev/LTest
[3] https://frama-c.com
[4] https://wookey-project.github.io/target.html
[5] ITSEFs (Information Technology Security Evaluation Facility), or CESTIs in French
[6] https://github.com/acsl-language/acsl/releases/latest

an Acsl specification, as shown in Fig. 4. The implementation of this function is not required for our technique based on deductive verification. We can simulate at most k faults (for any given $k \geq 1$) by changing the `MAX_MUTATION` macro definition to k (here, we set $k = 1$). The specification on lines 3–13 guarantees that `mutated` will return true (i.e. nonzero) at most `MAX_MUTATION` times during an execution. The order in which the true and false values are returned is left unspecified: it can be arbitrary. To count how many times the function has returned true so far, we introduce a new variable `cpt_mut`. Line 4 specifies that `mutated` only modifies `cpt_mut`, so it does not interfere with the application code. Then, two cases are considered (lines 5 and 9). If the maximal number of mutations has been hit (line 6), the function returns false, i.e. does not trigger a mutation, and the counter is unchanged (lines 7–8). If the maximal number of mutations is not reached (line 10), then the function either returns true and increments the counter (i.e. triggers a mutation), or returns false and leaves the counter unchanged (lines 11–12).

As shown in Fig. 3b (e.g. line 1), for each test of a condition C_i, a call to `mutated` is added and its result is stored in a variable `mut_i`, that we call a *mutation trigger*. The role of mutation trigger `mut_i` is to indicate if a test inversion should be performed on the condition C_i.

Then the value of C_i is combined with the mutation trigger in order to trigger a test inversion, i.e. to take the opposite branch, if `mutated` returned true (e.g. line 2). Such instrumentation can be compared, albeit at a slightly higher level of abstraction, to the one done by Lazart [12] over LLVM bitcode or by [11] over a translation of assembly code at C level.

Verified Properties. At the end of the critical code segment (line 8 in Fig. 3b), we insert a `check` annotation, which states that all mutation triggers are false at the protected point. If this `check` is proved, we can conclude that the protected code section can never be reached through up to k test inversions, or in other words, that redundant-check countermeasures are correctly implemented. Indeed, if there is an attack path with a nonzero number of faults, i.e. on which some mutation triggers are true, this annotation cannot be proved in general. To try to prove this annotation, we rely on the Wp plugin of Frama-C [9], that is based on deductive verification.

One Difficulty: Function Calls and Loops. In the presence of function calls and loops, deductive verification tools like Wp usually rely on manually written specifications. In our approach, having to provide them would come against our goal to make the verification process as automated as possible. For instance, to deal with the example of Fig. 2, the user would need to provide a complete specification of function `check_code_integrity`. To avoid these pitfalls, we propose to inline the called functions and unroll the loops so that the deductive verification tool can reason about the code without additional annotations. This solution has limitations: it will not work, for example, if the maximal number of loop iterations is very large or cannot be bounded. In practice, in critical code segments, loops with unbounded number of iterations are not so common, and it is often possible to determine the maximal number of loop iterations (a password is read at most three times, the secret code to check or the payload to copy is of a fixed ou bounded finite length, etc.). In our case study, this solution indeed allowed avoiding additional specifications, except in one case (see below).

Critical Section Location (Start-End)	Contains Loops	Contains Fun. Call	Nb of Object. (Unrolled)	Proof Result	Analysis
automaton.c:61-65	no	no	1 (13)	✓	Correct
automaton.c:368-374	no	no	1	✗	Bug
automaton.c:404-407	yes	yes	1	✓	Correct
automaton.c:426-429	yes	yes	1	✓	Correct
hash.c:86-91	no	no	1	✓	Correct
hash.c:114-122	yes	yes	1	✗	Correct?
main.c:408-453	no	no	2	✓	Correct
main.c:418-428	no	no	2	✓	Correct
main.c:429-439	no	no	2	✓	Correct
main.c:455-476	no	no	2	✓	Correct
main.c:569-578	no	no	1	✓	Correct
Total (before unrolling/inlining)			11		
Total (after unrolling/inlining)			27		
Proved			25/27		
Time			130s		

Figure 5: Summary of experiments with LTest on the countermeasures in WooKey

3 EXPERIMENTS

The approach described above has been implemented in LTest [3], a set of tools for coverage-oriented testing, mostly written in OCaml as plugins of Frama-C [9], a program analysis platform for C code. One of the plugins, LAnnotate, creates test objectives for given criteria. We implemented our technique as a new criterion, Redundant Check Countermeasures (RCC), which instruments countermeasures using annotations provided by the user. The instrumented code can then be given to another plugin, LUncov, which tries to prove, using Wp, that the target point cannot be reached by N test inversions.

We evaluated our tool on the 11 critical sections with redundant-check based countermeasures in the bootloader of WooKey[7]. Parts of code that are not protected by countermeasures were not considered. The table in Fig. 5 gives an overview of our experiments on all redundant check countermeasures in WooKey. For each critical section protected by redundant-check based countermeasures, the first columns provide the location of the section, whether it contains loops or function calls, the number of resulting objectives (that is, assertions to prove). When different, we also give the number of objectives after loop unfolding and function inlining. This number increases for the first section since the assertion is located inside the body of the unrolled loop. (In other cases, the number of assertions does not change despite using loop unrolling or function inlining.) The last two columns show the result of the proof (on the initial code) and our manual analysis of this result.

We were able to prove the efficiency of 9 of them, in around 2 minutes. One unproven section is correct but too hard to prove using Wp without adding function contracts and loop invariants: it has three function calls involving loops and bitwise operations, so that complete unrolling and inlining generates complex verification

---

[7] The tools were applied on the version of commit 00fd1c6 available on https://github.com/wookey-project/bootloader/.

```
1   /* double if protection */
2   if (new_state == 0xff && !(new_state != 0xff)) {
3       dbg_log("%s:␣PANIC!␣this␣should␣never␣arise␣!",
            __func__);
4       dbg_flush();
5       loader_set_state(LOADER_ERROR);
6       return;
7   }
8   //Safe code
```

Figure 6: Wrong countermeasures in WooKey

```
/* double if protection */
if (C1 && C1) {
    // Safe code
}
// Error
```

(a) Safe code inside

```
/* double if protection */
if (C1 || C1) {
    // Error
}
// Safe code
```

(b) Safe code outside

Figure 7: Good countermeasure patterns

conditions. Its proof using additional annotations is left as future work.

The second unproven section is shown in Fig. 6. The code sets the bootloader state to `LOADER_ERROR` and exits if the value of `new_state` is erroneous. The countermeasure is created by taking the conjunction of the original condition and its equivalent reformulation (see line 2). Using our method, we were not able to prove this countermeasure to be effective. In this case, it is indeed erroneous: the doubled condition on line 2 does the opposite of its purpose. Instead of protecting the critical code section, it allows a single mutation to bypass the check. Figure 7 gives correct patterns for doubled conditions (where the occurrences of condition C_1 can be equivalent but not necessarily the same, notably to avoid side-channel indications of redundant-check locations). The error here comes from the use of the logical operator AND (&&) instead of OR (||), unlike in Fig. 7b. This type of countermeasure, using a logical connector instead of successive `if` statements, occurs a few times in the WooKey bootloader, but this is the only case where the protected code is outside the condition block. In all other cases, the error is caught after, like in Fig. 7a.

This example shows that it is very easy to make an error in countermeasures that is not easy to find, as the code still works. It confirms the need for dedicated tools for verifying the countermeasures. After correcting the detected error, we proved a correct implementation of this redundant-check based countermeasure as well. Thus, *our tool was able to automatically prove ~90% of critical sections in the target module without adding contracts.*

4 CONCLUSION

We have proposed in this paper a method for formally verifying at source-code level that countermeasures against test inversion attacks are correctly implemented and shown that it can be successfully used on real C code thanks to Frama-C/LTest. We believe that providing a suitable implementation for the `mutated` function will make our instrumentation suitable also for finding faults using a test generation based approach (like Lazart [12]). If so, it will allow the engineer to use the same instrumentation both to formally prove the correct implementation of countermeasures and to generate test cases illustrating attacks. This extension is part of future work. Future work also includes considering other fault models, notably allowing the attacker to flip the value of any variable (up to k times) during the execution, automating the approach of [7] thanks to the use of LAnnotate.

Acknowledgements. This work was partially supported by ANR project SATOCROSS (grant ANR-18-CE25-0015-01). The authors thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] ANSII and French ITSEFs. 2020. Inter-CESTI: Methodological and Technical Feedbacks on Hardware Devices Evaluations. In *SSTIC Symposium*. 105–200. https://actes.sstic.org/SSTIC20/sstic-2020-actes.pdf

[2] ANSSI. 2021. Règles de programmation pour le développement sécurisé de logiciels en langage C.

[3] Sébastien Bardin, Omar Chebaro, Mickaël Delahaye, and Nikolai Kosmatov. 2014. An All-in-One Toolkit for Automated White-Box Testing. In *Proc. of the 8th International Conference on Tests and Proofs (TAP 2014)*. Springer, 53–60. https://doi.org/10.1007/978-3-319-09099-3_4

[4] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. 2012. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proc. IEEE* 100, 11 (2012), 3056–3076. https://doi.org/10.1109/JPROC.2012.2188769

[5] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. 2010. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proc. of the 5th Workshop on Embedded Systems Security (WESS 2010)*. ACM. https://doi.org/10.1145/1873548.1873555

[6] Shivam Bhasin, Paolo Maistri, and Francesco Regazzoni. 2014. Malicious wave: A survey on actively tampering using electromagnetic glitch. In *Proc of the 2014 Int. Symp. on Electromagnetic Compatibility*. IEEE, 318–321. https://ieeexplore.ieee.org/document/6997169

[7] Maria Christofi, Boutheina Chetali, Louis Goubin, and David Vigilant. 2013. Formal verification of a CRT-RSA implementation against fault attacks. *J. Cryptogr. Eng.* 3, 3 (2013), 157–167. https://doi.org/10.1007/s13389-013-0049-3

[8] Karine Heydemann, Jean-François Lalande, and Pascal Berthomé. 2019. Formally verified software countermeasures for control-flow integrity of smart card C code. *Comput. Secur.* 85 (2019), 202–224. https://doi.org/10.1016/j.cose.2019.05.004

[9] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27, 3 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

[10] Guilhem Lacombe, David Féliot, Etienne Boespflug, and Marie-Laure Potet. 2021. Combining Static Analysis and Dynamic Symbolic Execution in a Toolchain to detect Fault Injection Vulnerabilities. In *Proc. of the 10th International Workshop on Security Proofs for Embedded Systems (PROOFS 2021), colocated with the 2021 Conference on Cryptographic Hardware and Embedded Systems (CHES 2021)*.

[11] Johan Laurent, Christophe Deleuze, Vincent Beroulle, and Florian Pebay-Peyroula. 2019. Analyzing Software Security Against Complex Fault Models with Frama-C Value Analysis. In *Proc. of the 2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2019)*. IEEE, 33–40. https://doi.org/10.1109/FDTC.2019.00013

[12] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. 2014. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Proc. of the 7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*. IEEE, 213–222. https://doi.org/10.1109/ICST.2014.34