Verifying Redundant-Check Based Countermeasures: A Case Study

MFS 2022, Fréjus, March 21-23, 2022

Thibault Martin\textsuperscript{1}, Nikolai Kosmatov\textsuperscript{1,2}, Virgile Prevosto\textsuperscript{1}

\textsuperscript{1}Université Paris-Saclay, CEA, List, Palaiseau, France
firstname.lastname@cea.fr
\textsuperscript{2}Thales Research & Technology, Palaiseau, France
nikolaikosmatov@gmail.com
Outline

Fault Injection and Countermeasures

Verification Approach

Difficulties: Function calls and loops

Implementation

\texttt{ WooKey } Case Study

Conclusion and Future Work
Fault Injection and Countermeasures
Fault Injection
Fault model: test inversion, a very useful model [ANSSI & Inter-ITSEF, SSTIC’20]

- Attacker can invert up to $k$ arbitrary tests (checks) in the code (for a given $k \geq 0$)
- It is unlikely to inject $k + 1$ faults in a coordinated way

Countermeasure: redundancy of checks for critical conditions

- repeat (possibly, rewritten) critical checks at least $k + 1$ times each

Example: for $k = 1$, a password check is repeated twice

- If attackers bypass one check, the redundant check still prevents access.

```plaintext
if(password != secret) return 1;
if(password != secret) return 1;
// Protected area
```
Fault model: test inversion, a very useful model [ANSSI & Inter-ITSEF, SSTIC’20]

- Attacker can invert up to $k$ arbitrary tests (checks) in the code (for a given $k \geq 0$)
- It is unlikely to inject $k + 1$ faults in a coordinated way

Countermeasure: redundancy of checks for critical conditions

- repeat (possibly, rewritten) critical checks at least $k + 1$ times each

Example: for $k = 1$, a password check is repeated twice

- If attackers bypass one check, the redundant check still prevents access.

```c
if(password != secret) return 1;
if(password != secret) return 1;
// Protected area
```
Fault model: test inversion, a very useful model [ANSSI & Inter-ITSEF, SSTIC’20]

- Attacker can invert up to $k$ arbitrary tests (checks) in the code (for a given $k \geq 0$)
- It is unlikely to inject $k + 1$ faults in a coordinated way

Countermeasure: redundancy of checks for critical conditions

- repeat (possibly, rewritten) critical checks at least $k + 1$ times each

Example: for $k = 1$, a password check is repeated twice

- If attackers bypass one check, the redundant check still prevents access.

```c
if(password != secret) return 1;
if(password == secret) return 1;
// Protected area
```
How to ensure that countermeasures are correctly implemented?
Verification Approach
Find patterns and delimit critical code

The user needs to delimit the critical section(s) and to identify the protected point(s)

```c
// Critical zone start
if(C_1) return 1; // Error
...
if(C_N) return 1; // Error
// Protected area

// Critical zone end
```
Instrumentation to Simulate Faults

// Critical zone start
if(C1)
    return 1;
...
if(CN)
    return 1;
// Protected area
// Critical zone end

```c
int mut_1 = mutated();
if(!mut_1 && C1 || (mut_1 && !C1))
    return 1;
...
int mut_N = mutated();
if(!mut_N && CN || (mut_N && !CN))
    return 1;
/*@ check !mut_1 && ... && !mut_N;*/
// Protected area
```
**Instrumentation to Simulate Faults**

- `mut_i` represents a mutation trigger for $C_i$.
- `mutated()` returns `true` at most $k$ times non-deterministically.
- The assertion states that the protected area can never be entered after a mutation (i.e., an attack).

```c
int mut_1 = mutated();
if((!mut_1 && C_1) || (mut_1 && !C_1))
    return 1;
...

int mut_N = mutated();
if((!mut_N && C_N) || (mut_N && !C_N))
    return 1;

/*@ check !mut_1 && ... && !mut_N;*/
//@ Protected
```
Prove Assertions Using Automatic Tools

- Apply deductive verification
- Try to prove the check annotation

```c
int mut_1 = mutated();
if(!(mut_1 && C1) || (mut_1 && !C1))
    return 1;
...
int mut_N = mutated();
if(!(mut_N && C_N) || (mut_N && !C_N))
    return 1;
/*@ check !mut_1 && ... && !mut_N */
//Protected
```

If the check annotation is proved, the specified critical section is correctly protected.
unsigned int cpt_mut = 0;

/*@
   assigns cpt_mut;
   behavior cannot_mutate:
   assumes cpt_mut >= k;
   ensures !result;
   ensures cpt_mut = \at(cpt_mut, Pre);
   behavior can_mutate:
   assumes cpt_mut < k;
   ensures result <=> cpt_mut = \at(cpt_mut, Pre) + 1;
   ensures !result <=> cpt_mut = \at(cpt_mut, Pre);
*/

int mutated();
Difficulties: Function calls and loops
Deductive Verification without Annotations?

Weakest precondition is in general well-adapted for local reasoning but can face issues:

<table>
<thead>
<tr>
<th>Function calls</th>
<th>Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>- How to avoid the need to write function contracts?</td>
<td>- How to avoid writing loop contracts?</td>
</tr>
<tr>
<td>➤ Inline called functions</td>
<td>➤ Use loop unrolling</td>
</tr>
<tr>
<td>➤ It can make proof complex</td>
<td>➤ It can duplicate critical areas</td>
</tr>
<tr>
<td>➤ It can introduce loops</td>
<td>➤ Loop bounds can be high / unknown</td>
</tr>
</tbody>
</table>
// Critical area starts here?
int i = 0;
while (i < SIZE) {
    // or starts here?
    if (password[i] != secret[i]) return 1;
    if (password[i] != secret[i]) return 1;
    ...
    i++;
    // Critical area ends here?
}
// or ends here?
Vulnerability of Non-protected Loop Conditions

// Critical area start
int i = 0;
while(1) {
    // Injection here can lead to undefined behavior
    if(!(i < SIZE)) break;
    if(password[i] != secret[i]) return 1;
    if(password[i] != secret[i]) return 1;
    ...
    i++;
}
// Critical area end
// Critical area start
int i = 0;
while(i < SIZE){
    if(!(i < SIZE)) break;
    if(password[i] != secret[i]) return 1;
    if(password[i] != secret[i]) return 1;
    ...
    i++;
}
// Critical area end
• Redundant-check countermeasures can follow different kinds of patterns
• We need to know which pattern is used to perform a correct instrumentation

```java
if(password != secret) return 1;
if(password != secret) return 1;
if(password != secret || password != secret) return 1;
```
Implementation
Frama-C: a platform for C program verification developed by CEA List

Frama-C/Wp: weakest precondition based tool for deductive verification

Frama-C/LTest: toolset for program testing
  - LAnnotate: performs code instrumentation
  - LUncover: Calls Wp to attempt a proof of assertions

If all assertions are proved, specified critical sections are correctly protected
WooKey Case Study
WooKey: A secure USB Mass Storage

- Open-source and open-hardware
- Developed by the ANSSI
- Secured by data encryption
Bootloader

- Select boot mode
- Select one of two boot areas
- CRC and Integrity check
- Boot
Example of Countermeasure in WooKey

Several countermeasures with code redundancies

/* Double sanity check (for faults) */
if(fw->fw_sig.len > partition_size){
    goto err;
}
if(fw->fw_sig.len > partition_size){
    goto err;
}
Results of Experiments

- 11 critical sections using countermeasures
- 3 involving loops and function calls
- 9 proved correct
- 1 cannot be proved without annotations (probably correct)
- 1 incorrect countermeasure found (and proved after fixing)
Incorrect Countermeasure: No Protection

```c
/* Duplicated check */
if (new_state == 0xff && !(new_state != 0xff)) {
    dbg_log("%s: PANIC! this should never arise!", __func__);
    dbg_flush();
    loader_set_state(LOADER_ERROR);
    return;
}
// Safe code
```
Incorrect Countermeasure: No Protection

/* Duplicated check */

if (new_state == 0xff && !(new_state != 0xff)) {
    dbg_log("%s: PANIC! this should never arise!", __func__);
    dbg_flush();
    loader_set_state(LOADER_ERROR);
    return;
}

// Safe code
Correct Implementation: Protection is Ensured

/* Duplicated check */
if (new_state == 0xff || !(new_state != 0xff)) {
    dbg_log("%s: PANIC! this should never arise!", __func__);
    dbg_flush();
    loader_set_state(LOADER_ERROR);
    return;
}

//Safe code
Correct Implementation: Protection is Ensured

/* Duplicated check */
if (new_state == 0xff || !(new_state != 0xff)) {
    dbg_log("%s: PANIC! this should never arise!", __func__);
    dbg_flush();
    loader_set_state(LOADER_ERROR);
    return;
}

//Safe code
Pay attention to what you are trying to protect:

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

```c
/* Double if protection */
if (C1 || C1) {
    // Error
}
// Safe code
```
Conclusion and Future Work
Summary

- A new method to prove correctness of redundant-check countermeasures
- Implemented in Frama-C and LTest
- Successfully applied to a real case study: WooKey
  - Automatically proved 90% of countermeasures
  - Helped to find an incorrect one

For more detail, see [Martin et al, SAC-SVT’22]

What’s next?

- Other experiments and case studies
- Combine with tools for attack generation