Compiler-Assisted Loop Hardening Against Fault Attacks

JULIEN PROY, INVIA, France
KARINE HEYDEMANN, Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6, Paris, France
ALEXANDRE BERZATI, INVIA, France
ALBERT COHEN, INRIA and DI, École Normale Supérieure, Paris, France

Secure elements widely used in smartphones, digital consumer electronics, and payment systems are subject to fault attacks. To thwart such attacks, software protections are manually inserted requiring experts and time. The explosion of the Internet of Things (IoT) in home, business, and public spaces motivates the hardening of a wider class of applications and the need to offer security solutions to non-experts. This article addresses the automated protection of loops at compilation time, covering the widest range of control- and data-flow patterns, in both shape and complexity. The security property we consider is that a sensitive loop must always perform the expected number of iterations; otherwise, an attack must be reported. We propose a generic compile-time loop hardening scheme based on the duplication of termination conditions and of the computations involved in the evaluation of such conditions. We also investigate how to preserve the security property along the compilation flow while enabling aggressive optimizations. We implemented this algorithm in LLVM 4.0 at the Intermediate Representation (IR) level in the backend. On average, the compiler automatically hardens 95% of the sensitive loops of typical security benchmarks, and 98% of these loops are shown to be robust to simulated faults. Performance and code size overhead remain quite affordable, at 12.5% and 14%, respectively.

CCS Concepts: • **Security and privacy → Hardware attacks and countermeasures**; **Software and application security**; • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Compiler, software protection, physical attacks

ACM Reference format:
Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. 2017. Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Trans. Archit. Code Optim.* 14, 4, Article 36 (December 2017), 25 pages.
https://doi.org/10.1145/3141234

1 INTRODUCTION

Sensitive data is no longer limited to secure devices such as passports for authentication or smart cards for payment. These devices are designed for resiliency against physical attacks; such protections are mandatory to satisfy the requirements of certification authorities. With the emergence of the Internet of Things (IoT), personal data may be handled by a huge variety of devices, escaping certification regulations and generally hardened with ad-hoc solutions. These systems are also subject to physical attacks, among which are fault attacks that aim at disrupting the execution of some applications to extract sensitive information [11] or grant restricted access permissions [18,

Authors' addresses: J. Proy and A. Berzati, INVIA, Arteparc Bât D., route de la côte d'azur, 13590 MEYREUIL, France; emails: {julien.proy, alexandre.berzati}@invia.fr; K. Heydemann, LIP6, 4 Place Jussieu, 75252 Paris Cedex05, France; email: karine.heydemann@lip6.fr; A. Cohen, DI, Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France; email: albert.cohen@inria.fr.

https://doi.org/10.1145/3141234

47]. In the last decade, different methods have been tested to inject faults, such as electromagnetic or lazer radiation, and power or clock signal tampering [4, 9], with diverse effects on the device under attack [4].

As a reaction, countermeasures have been proposed to protect embedded systems against faults, both in hardware and in software. To provide maximum security, it is necessary to combine both of them, as hardware and software protections tend to complement each other. The current practice in industry is to insert protections manually or to activate specific hardware mechanisms, often at the assembly or binary level. This does not scale and impacts the overall development costs. It relies on experts' knowledge and is also error-prone, as it is very difficult to anticipate all interactions between control paths and the different threats, or to manage their combinatorial complexity. Moreover, the diversity of devices providing secure services makes it necessary to harden a wider class of applications and to offer security solutions to non-experts. As a result, there is a growing need for tools able to automatically insert software protections. Since most candidate devices are small, and their recurring production costs push for maximal efficiency, code hardening must be performed while considering other concerns such as code size, memory footprint, and performance.

Interestingly, since automation implies raising the level of abstraction of security protections, it reduces the risk of misinterpreting non-portable, optimized protection code over software maintenance and evolution cycles. Code hardening can be performed at source level [21, 28], in dedicated compilation passes [2, 6], or at link-time, possibly involving a binary rewriting tool [14].

- Source-to-source approaches are conditioned to the effects of the downstream compilation passes. These may damage or even completely remove the high-level protections, since many countermeasures involve duplicated computations classified as redundant expressions by compiler optimizations. Moreover, there is no one-to-one correspondence between the models of attack considered at source level and the observed effects of fault injection at the (micro-) architectural level. As a consequence, source-to-source approaches do not preclude analyzing the generated assembly code, and they often require disabling optimizations of the hardened procedures to guarantee the protection's effectiveness.
- Binary rewriting approaches eliminate the time-consuming, trial-and-error process of applying source-level protections. Their application at link-time or post-link development cycles offer more agility in secure software development and maintenance. Yet binary rewriting involves the recovery of a higher level representation from low-level code with analyses whose semantical reach of complex program properties remains limited. It also involves the reservation of many registers to deploy the dedicated protection code, involving extra overhead [14].
- Automatic code hardening at compilation time is more attractive: protection deployment is made transparent to the developers, improving their productivity and avoiding the duplication of efforts at multiple levels of the program representation. Developers may concentrate on the functional correctness of the application, and the compiler may issue a warning in case the enforcement of the security property could not be validated automatically. Moreover, compilation-time hardening can take advantage of code optimization without the drawbacks of binary rewriting.

However compilation-time hardening comes with its own challenges. First of all, each security property needs to be properly enforced by a code transformation. The design and implementation of these require generalizing existing practices from the folklore of tricks used by expert security developers and from published schemes that typically target a small class of applications like block-cipher kernels [43]. Against fault attacks, broader schemes have been proposed. As an example, instruction-level countermeasures [32] define a replacement sequence potentially followed by a

comparison to detect a possible disruption [5]. Such narrow-scope protections are easily automated at compilation time [6], yet they induce a significant performance and code size overhead that is not compatible with embedded constraints. Moreover, they do not capture higher level information such as loop invariants or termination post-conditions. In general, the implementation of a wider range of security properties requires wider-scope protections such as loop-level countermeasures.

The most common property to be secured at loop level is the enforcement of a correct iteration count. For the Advanced Encryption Standard (AES) cipher, it has been shown that corrupting the loop counter enables to retrieve the key [15]. More generally for cryptographic applications, their mathematical properties cannot be guaranteed if the appropriate number of iterations is either lower or higher than the expected one, making them vulnerable to cryptanalysis [7, 16, 20, 34, 36, 42]. At system level, recent work has shown the practicality of fault injections in the core loop of the C library `strncpy` function, resulting in a harmful buffer overflow [37]. Other work highlighted the need to protect the iteration count of the pin code verification of smart cards [18]. Simple cases of loop counter protection have recently been handled at binary level [14]. As loops may have very diverse control flow (e.g., possibly unwinding the termination condition, with multiple exit points), with deep nesting and complex termination conditions, more advanced protection schemes are needed. The application of such protections, in turn, calls for hardening algorithms operating on the compiler intermediate representation, where rich data and control flow information is available. As a result, compilation-time hardening approaches raise a difficult *robustness* question: how can the compiler ensure that the security property applied to the intermediate representation will still protect the generated assembly code? Compiler optimizations for performance or code size are designed to preserve the observable semantics of the input program. Yet countermeasures against physical attacks typically rely on some redundancy in the control and data flow of the program. Such redundant operations do not impact the observable semantics; they are tracked and removed by any optimizing compiler through dead code elimination, code motion, common sub-expression elimination, global value numbering, induction variable canonicalization, and the like [35]. The elimination of post-pass assembly code analyses and of the associated trial-and-error process calls for an integrated solution implemented in the compiler. We do not foresee that such a problem will find a general, compiler-independent answer. Instead, we propose a hardening algorithm operating on the intermediate representation of a specific compiler, together with the methodology, to embed this algorithm into a more generic optimizing compilation flow. The positioning of the hardening pass results from a careful analysis of the compilation flow, considering both its generic intermediate representation and target-specific backend. It reflects the dependence on upstream passes providing the necessary semantical information to implement the countermeasure and on interferences caused by downstream passes. The methodology addresses this difficult tradeoff, minimizing the changes to the downstream compilation flow and maximizing the applicability of aggressive optimizations on the secured code. Our main technical contribution consists in a generic loop hardening scheme as well as a compilation-time protection pass to deploy it. The security property we consider is that, during a given run of the application, any sensitive loop must perform the expected number of iterations and take the expected exit path, under a well-defined attack model. The protection pass handles general and complex control and data flow. To take full advantage of the compiler's aggressive optimizations and for portability consideration, we implement this pass at the level of the intermediate representation, benefiting from high-level semantical properties while enforcing the robustness of the inserted countermeasure through the compilation flow. This is made possible by the analysis of interactions with upstream and downstream passes in the compiler, including optimizations that might interfere with it. Experimental results show that our method is able to harden 95% of the sensitive loops, detecting 98% of the simulated faults, and incurring only a small performance and code size overhead compared

to more generic protection schemes. This article is organized as follows: Section 2 introduces physical attacks and the associated attack model. Section 3 motivates our hardening scheme. Section 4 follows with pass ordering and compiler construction, and Section 5 presents experimental results. Related work is discussed in Section 6 before the concluding statements in Section 7.

2 BACKGROUND AND MOTIVATING EXAMPLE

Numerous papers have demonstrated that physical attacks are a powerful way to retrieve secret information or escalate privileges on secure systems. Let us first position our work with respect to the state-of-the-art threat and attack models in this area.

2.1 Physical Attacks

One typically divides physical attacks in two classes:

- Side-Channel Attacks (SCAs) exploit correlations between physical measurements (time, electromagnetic radiations, power consumption, etc.) and some secret information targeted by the attacker [27]. SCAs are also referred to as passive attacks since they can be set up without altering the target device.
- Fault Attacks (FAs) induce malicious alteration of the device to retrieve some sensitive data, to get privileged access, or to bypass protections. These attacks are invasive since the target device is typically irradiated to trigger the faulty behavior (e.g., light, electromagnetic probe). The induced faults are generally transient: most of the time, the device recovers its original behavior once the source of the fault is turned off.

This article focuses on fault attacks. Injecting a fault is not straightforward: the attacker has to control—even partially—the effects of the fault so that it has a reasonable chance to expose sensitive information. To do so, the attacker must gain access to specialized equipment and follow cumbersome procedures to prepare the device [4]. Moreover, a single type of fault may be triggered by different means [25], with varying effects [3, 4, 46]. Yet previous work demonstrated that exposing a chip to laser pulses induces most of the effects described in the literature:

- single or multiple bits may be set or reset in a register, which can be achieved forcing the value of a single or multiple bits to `0` or `1` by respectively faulting the reset or set signal of a flip-flop [4, 10, 11, 41];
- single or multiple bit flips in a register, which is achieved by switching the state of a memory cell or flip-flop [4, 19, 40];
- random register modifications that extend the previous model to different word size (e.g., from 8 to 64 bits);
- random alteration of transfers between the CPU and dynamic or non-volatile memory [31];
- instruction replacement by instruction fetch corruption [31].

These effects can be exploited to fool a sensitive computation and take over the device. To reduce sensitivity to fault attacks, hardware protections can give information about the processor state or implement detectors that trigger an alarm in case of unexpected conditions (e.g., temperature or light sensors). Nevertheless, hardware protections may still be bypassed, and sensitive functions must still embed software countermeasures to deflect or detect the attempts of a malicious attacker.

2.2 Fault Models

Software protections against fault attacks are designed with respect to fault models that define the exploited weaknesses and the effects of fault injections at the code representation of interest. In this article, we consider the following fault model:

```
int i = n, j = n;
while (i != 0) {
  foo(i);
  if (Array[i] == chkVal) break;
  i--; j--;
}
if (j != 0 && Array[j] != chkVal)
  error();
```

Listing 1. Multiple exists.

```
int r = a%b, r' = a%b;
while (r != 0) {
  a = b;
  b = r;
  r = a%b, r' = a%b;
}
if (r' != 0)
  error();
```

Listing 2. Complex induction.

```
int i = n;
while (i > 0) {
  if (i == 5) {
    i -= 2; /* ... */
  } else {
    i--; /* ... */
  }
}
```

Listing 3. Conditional induction.

— *Instruction skip*: the instruction executed is not the expected one and has no effect; it is a specific case of instruction replacement but the most frequent in practice [31]. Note that the probability an attacker injects a specific replacement with an observable effect is very low and depends on the Instruction Set Architecture (ISA) and instruction encoding.
— *Random register corruption*: a general-purpose register is randomly corrupted, excluding the program counter. Considering the observable effects of physical attacks described in the previous section, our model covers single and multiple bits being forced or flipped. Considering laser injections, this model is considered to be one of the most practical for an attacker.

Note that data transfer to and from memory, from the CPU side, is also covered by our model, since the register holding the result of a load or the register holding the data to be stored can be corrupted. From the memory side, effects are harder to model since data are usually protected by consistency checks (e.g., error-correcting codes for non-volatile memories). Moreover, on secure devices, the memory contents is usually encrypted, which makes it hard to recover logical fault effects from the physical ones. These security mechanisms complement our approach to maximize the robustness of a component.

Recent works have demonstrated the possibility to inject multiple faults during the execution of a cryptographic algorithm [17, 30, 49]. Since the first demonstrations and exploits [26], the practicability of the techniques has improved, leading security certification agencies to integrate it in their evaluation process [12]. The main purpose of multiple fault attacks is to bypass security mechanisms, and at the same time, faulting the computations of interest so that the attacker can get access to the faulty result and exploit it. Most of the time, algorithmic security mechanisms to detect faults are based on executing the inverse operation on the output (e.g., decryption or signature verification) and comparing to the expected result. The reader may notice that these extra operations involve loops, which emphasize the importance of our motivating scenarios. In this article, we consider only one single fault per loop; this matches to the most likely scenario, considering that in presence of multiple faults, it is hard to control them because they specifically hit the same loop instance. Each individual loop is being protected individually, considering a single fault per loop is sufficient to cover multiple faults as defined by the literature and certification documents.

2.3 Loop Hardening

As previously explained, tampering with loop iteration count can leak sensitive data or give control to an attacker. To the best of our knowledge, there exists no generic hardening scheme for the loop iteration count. Control-flow integrity measures do not track the data flow leading to a specific branch: they may detect whether a loop is entered or exited, but not how many iterations it takes [28]. We will also show that loop hardening does not incur the large overheads of general-purpose, fine-grained control-flow integrity.

The common practice is to introduce additional variables to harden the loop iteration count [18]: on the example in Listing 1, the iteration counter is duplicated as well as the computation of the exit condition, forcing a second check at loop exit—the code implementing the countermeasure is highlighted in red-colored italics and it double-checks that the exit condition holds. This protection is a widespread form of temporal duplication followed by a detection step, with the granularity set to the level of an individual instruction rather than a full function.

This is a simple although typical example, where the countermeasure is deployed at source code level: any optimizing compiler will remove the extra variable `j` since it is a copy of the iteration counter `i`. The result is equivalent to only duplicating the computation of the exit condition and the out-of-loop exit check using variable `i`. Unfortunately, this is no more robust against faults that corrupt the loop exit condition. Besides, depending on the optimization level, the compiler is even able to entirely remove the extra check outside the loop: from a compiler view, if the first one holds, the second one does, too. It does not take into account potential fault attacks. It may be possible to trick the compiler from removing the added code by declaring the extra variable *volatile*. However, it comes with a performance overhead, and such approaches are not sound since the developer still has to check the resulting assembly.

Moreover, the example in Listing 2 shows that more complex exit conditions cannot be protected by only duplicating variables: indeed, any fault inducing a skip on the `b = r;` assignment would corrupt both the original and the redundant variable and would not be detected.

Other issues arise when the loop body contains different execution paths that influence the iteration count or the loop exit. The exit condition `i>0` in Listing 3 depends on the conditional branches taken during previous iterations (`i--` or `i-=2`). Hence, it depends on the condition `i==5`, which must then be hardened, too. This example highlights the need to harden any control flow path in a loop body when it impacts the exit condition.

Overall, a sound protection scheme requires a deep analysis of both control and data flow leading to the exit condition. Of course, it is not realistic to manually apply such a scheme on larger codes. We propose a generic protection scheme and show how it can be applied at compilation time. A redundant exit condition checks the validity of every branch associated with loop iteration or exit; any fault attack during loop execution that affects the number of iterations or the exit path will be detected. The basis of this scheme is to duplicate any instruction of the loop body involved directly or indirectly in the computation of every exit condition.

3 COMPILE-TIME LOOP HARDENING

Let us now present the compilation algorithm to automatically protect against fault attacks targeting the loop iteration count, as described in Section 2.2.

3.1 Prerequisites

This algorithm operates on a low-level Intermediate Representation (IR). To simplify algorithms and schemes, we assume the IR is target-independent and in Static Single Assignment (SSA) form, as implemented in production compilers like GNU Compiler Collection (GCC)[1] and The LLVM Compiler Infrastructure (LLVM).[2] In this IR, basic blocks are a sequence of instructions ending with a jump, in the form of a return, an unconditional branch, a conditional branch, or a switch statement.

Iterative control flow varies widely, depending on the application, the domain constraints and usage, the programming language, and the compilation flow. Like most loop transformations, our

[1] https://gcc.gnu.org.
[2] http://llvm.org.

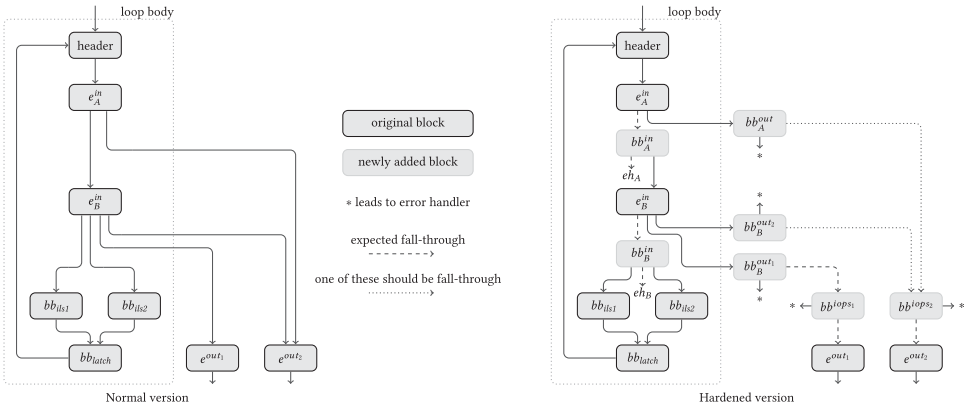

Fig. 1. Loop example with multiple exiting and exit blocks and its hardened version.

hardening algorithm expects a control flow analysis to construct a normalized representation of the nested loops in a function, such as the loop nesting forest [44]. Borrowing from LLVM terminology, the normalized form of a loop L is a tuple $(header, body, \{(e^{in}, e^{out})\})$ where:

— *header* is the unique basic block, which is a loop entry point;
— *body* is the list of basic blocks dominated by the header with a path leading back to it; it includes the header;
— the list $\{(e^{in}, e^{out})\}$ of exiting-exit block pairs contains all exiting edges of the loop.

An exiting block e^{in} belongs to the loop; it has at least one successor inside the loop and at least one successor outside the loop. Its jump instruction is then either a conditional branch or a switch. In case of a switch, an exiting block e^{in} can have multiple exit blocks $\{e^{out}\}_j$ as successors. Conversely, an exit block e^{out} does not belong to the loop, and it can have multiple exiting blocks $\{e^{in}\}_i$ as predecessors in the loop.

Figure 1 illustrates a loop with two exiting blocks e_A^{in} and e_B^{in}, three exiting edges, and two exit blocks e_1^{out} and e_2^{out}, and where e_B^{in} ends with a switch with two in-loop successors bb_{ils1} and bb_{ils2}.

Except in rare cases—such as an indirect branch into the loop or complex irreducible control flow—naturally occurring loops can be transformed to this form. In our experiments, we did not find any loop syntactically occurring in the C source that failed to be recognized.

3.2 Overview and Main Algorithm

The generic protection scheme we propose aims at hardening a sensitive loop against disruption of the iteration count or taking an incorrect exit. It duplicates, inside the loop, all instructions involved in the computation of all exit conditions and the associated loop-back logic. Any exit instruction is also duplicated outside the loop, before each of its associated exit blocks, to check the correct exit has been taken. Any failed check calls an error handler.

Analysis and transformations required to harden one loop are described in the top-level Algorithm 1. Figure 1 illustrates one loop example and its hardened version resulting from these transformations. Moreover, Appendix presents two examples with source, original, and hardened assembly code for ARMv7 M[3] ISA.

Algorithm 1 processes a loop by analyzing and transforming each of its exiting block e^{in}, then finalizes the hardening of every exit block e^{out}. For one exiting block e^{in}, the algorithm first

[3] http://www.arm.com.

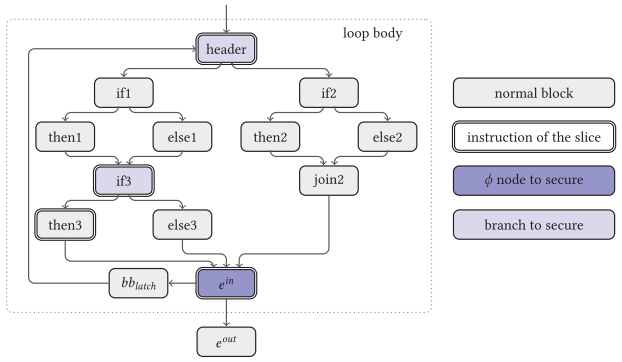

Fig. 2. Securing branches in the CFG.

retrieves the jump instruction *ei* of e^{in} and then the instruction *ci* that defines the condition which *ei* depends upon (line 4). This allows to determine instructions influencing the computation of this exit condition by computing a backward slice and the list *LBr* of all branch instructions leading to different in-loop paths that may influence its value. Such branches must be hardened, too, and *instToDup* collects all instructions on the backward slices computed from the condition of branch instruction in *LBr*. The list *LIops* of the encountered loop-independent operands is also updated during the analysis of e^{in} (line 6).

The transformation of one exiting block e^{in} consists in the following steps. Inside the loop, the transformation starts by duplicating all instructions from its slice (line 7). A block composed of a new conditional branch that depends on the duplicated exit condition is created and inserted inside the loop as a successor of e^{in} to check its outcome (line 10). An error handler block *eh* is also added to deal with detected disruptions. The hardening of branch instructions of *LBr* is handled by the secureBranches function (line 11). Then, for each exit block e^{out} successor of e^{in}, an additional basic block bb_{out} uses the duplicated exit condition to verify if this exit should have been taken; it is inserted in the Control Flow Graph (CFG) before e^{out} by the updateExitCondCFGout function (line 14).

Finally, dupLoopInvOps deals with the duplication of loop-independent operands from *LIops*, saving them before the first iteration in a block dominating the loop header. It creates a block bb_{iops} that compares their value to the stored ones, and ends with a global check that none of the values has been tampered with (line 15). For each exit block e^{out}, a copy of bb_{iops} is inserted in right before e^{out}, after all bb_{out} blocks (line 17). This adds a check of loop-invariant operands at this exit.

3.3 Stepping Inside the Loop Hardening Algorithm

Let us now detail the main steps of Algorithm 1.

Computation of the Backward Slice. The role of the getBackSlice function of Algorithm 1 is threefold: in a single traversal, it computes a backward slice of definition instructions and a list of in-loop branches to harden. It also updates the list of loop-independent variables; see Algorithm 1.

Starting from an exit condition, the analysis walks back through use-def chains (lines 17 to 22). At each step, when an in-loop definition instruction is dequeued from the work list and processed, it is added to the slice if not the case already (lines 5 and 6). The analysis will handle the in-loop definition of any operand of a newly encountered instruction by inserting it into the work list.

One such instruction can be a ϕ node. In SSA form, ϕ nodes are inserted at join points to select the incoming value according to the taken path and defining a fresh variable set to this value. As a result, the definition of a variable always dominates all its uses. Consider the example in Figure 2

ALGORITHM 1: Hardening Loop Iterations and Exits

```
1  L ← processLoop(L)
       Input: Loop L
       Output: Secured loop L
2      LIops = []
3      foreach exiting block e^in of L do
4          ei = getJump(e^in)
5          ci = getCondInst(e^in)
6          [instToDup, LBr, LIops] =
           getBackSlice(L, ci, LIops)
7          dupInsts(L, instToDup)
8          bb_in = newBlock(ei)
9          eh = newBlock()
10         updateCFGin(L, e^in, bb_in, eh)
11         secureBranches(L, LBr)
12         foreach exit block e^out successor of
           e^in do
13             bb_out = newBlock(ei)
14             updateExitCondCFGout(L, e^out,
               eh, bb_out)
15     bb_iops = dupLoopInvOps(L, LIops)
16     foreach exit block e^out do
17         updateIOpsCFGout(L, e^out, eh,
           dupBB(bb_iops))
```

ALGORITHM 2: Backward Slice and Loop-invariant Operands

```
1  [instToDup, LBr, LIops] ← getBackSlice(L, start, LIops)
       Input: Loop L; Instruction start; List of loop-invariant operands
              LIops
       Output: List of instructions instToDup; List of branches to secure
               LBr; List of loop-invariant operands LIops
2      worklist = [start]; instToDup = []; LBr = []
3      while worklist is not empty do
4          cur = dequeue(worklist)
5          if cur ∈ instToDup then continue
6          append(instToDup, cur)
7          if isPhiNode(cur) then
8              phiBB = getBB(cur)
9              foreach pair (Pred_i, Pred_j) ∈ inLoopPredsOf(phiBB) do
10                 if valueFrom(cur, Pred_i) =
                   valueFrom(cur, Pred_j) then
11                     continue
12                 ancs = relevantCommonAncestors(Pred_i, Pred_j)
13                 foreach a ∈ ancs do
14                     ei = getJump(a)
15                     appendIfNotListed(LBr, ei)
16                     appendIfNotListed(worklist, getCondInst(ei))
17         foreach operand next ∈ Operands(cur) do
18             if next is loop-independent then
19                 appendIfNotListed(LIops, next)
20                 continue
21             inst = getDefinitionInstOf(next)
22             if inst ∈ L then append(worklist, inst)
23     return [instToDup, LBr, LIops]
```

where instructions involved in the exit condition are only defined in blocks *header*, *if3*, *then3*, and e^{in}. Due to the definition of an operand in *then3*, which does not dominate e^{in}, the slice computed from the exit condition in e^{in} will include at least one ϕ node that belongs to e^{in}. To ensure the correct value is selected by such a ϕ node, it is mandatory to reach e^{in} from a legitimate path, i.e., from a predecessor whose defined value is the expected one. To ensure the execution comes from the expected predecessor, one can secure every branch leading to the ϕ node by starting from its immediate dominator (*header* in this example). While it is sufficient to consider all these branches, it is unnecessarily expensive. There is no need to secure a block with multiple outgoing paths that all lead only to the same predecessor of the ϕ node, or to a list of predecessors of the ϕ node feeding it the same value (block *if2* in the example):[4] any path from this block assigns the right value to the ϕ node. Hence, we only consider branches from common ancestors of two in-loop predecessors of a ϕ node that feed it with different values.[5]

However, among common ancestors of two predecessors, blocks that are post-dominated by another common ancestor do not require to be secured either. This is the case for the *if1* block in this example. Indeed, let us assume that faulting the branch of *if1* impacts the value received by the ϕ node. As every path from *if1* leads to *if3*, which is another (lower) common ancestor, the condition of *if3* must also be impacted on the path taken from *if1*. It means that the branch condition of *if3* depends on a ϕ node in the *if3* block, which in turn depends on the branch condition of *if1*. Thus, when considering the ϕ node of e^{in}, it is sufficient to secure only the branch of *if3*, since the ϕ nodes it depends upon will be encountered on the backward slice computed from its condition. All other branches to be hardened will then necessarily be encountered, inductively.

[4]Multiple predecessors may carry the same value in presence of ϕ nodes with three or more arguments, effectively silencing any fault swapping one path from the other.

[5]In case of ϕ node in the header, a predecessor can be outside the loop and thus not requiring attention.

Hence, for two predecessors of a ϕ node defining different values (line 9 in Algorithm 1), the branches to secure belong to paths from the closest common dominators of the predecessors (line 12) to the ϕ node itself that are not post-dominated by another common ancestor of the two predecessors.

Hence, the instruction defining the branch condition is added to the work list in order to continue the backward analysis from it (line 16) and the branch itself is added to *LBr* list to manage checking blocks later.

The backward dependence analysis stops when it meets a constant or loop-independent operand. In the latter case (lines 18 to 20), the operand is added to the *LIops* loop-invariant list (that list is used in the last step of the loop hardening scheme).

Instruction Duplication. To implement the actual duplication of instructions, the slice is sorted in topological order to preserve the causality of def-use chains when processing instructions sequentially. The main Algorithm (Algorithm 1) considers each exiting branch independently. However, some instructions in backward slices (Algorithm 1) can be common to several slices computed for different exiting blocks. Also, an instruction may have been already duplicated during the securing process of either another exiting of L or an inner or outer loop of L. Hence, during the duplication step, care is taken to avoid multiple replications of the same instruction.

Operands of each duplicate instruction are updated with the new ones by maintaining a renaming list. Duplicated instructions are placed just before the original ones in the loop body. This step processes all loop-dependent variables occurring into a given backward slice, effectively constructing a separate data-flow sub-graph isolated from the original loop-dependent variables.

Updating the CFG Inside the Loop. Two blocks are inserted into the loop for every exiting block e^{in}: the exit instruction *ei* is duplicated and inserted in a new block bb_{in}; the duplicate of *ei* is updated to replace every out-of-the-loop successor of e^{in} by a unique new block *eh* that calls an error handler.

Care must be taken in the placement of these inserted blocks, considering that skipping a branch leads to the execution of the fall-through block. The block bb_{in} should be placed as fall-through of e^{in} and *eh* as fall-through of bb_{in} itself. Thus, if a fault skips the branch of e^{in}, the execution will fall in the redundant checking block bb_{in}, and if a fault skips the branch of bb_{in}, it will fall into *eh*.

Branch Hardening Inside the Loop. For each successor bb_s of *br*, a duplicate of *br* is inserted in a new block. This duplicate branch uses the duplicated condition from the slice, and all its successors should be replaced by the error handler block *eh* except for bb_s.

This block is then inserted right before bb_s such that if a skip of the branch *br* causes the execution to fall into bb_s, execution will proceed with this new checking block.

Duplication of Loop-invariant Values. During the slice computation for all exit conditions, some instructions might have loop-independent operands that are put in a single special list *LIops*. The duplicate of these instructions use the same loop-invariant operand. However, the loop hardening scheme must detect any loop-independent values' corruption.

Each loop-independent operand could be duplicated and used in duplicated instructions. However, it would increase the number of live variables and register pressure. Moreover, these copied operands can be captured by peephole passes in the backend that do not require full-function data flow information. So, our practical solution is to effectively store each loop-invariant operand in a dominant block of the header (for instance, the function entry) into a dedicated location. Hence, instructions and their duplicate use the same loop-invariant operands.

To detect any corruption of these operands during loop execution, a new basic block bb_{iops} is created. In this new block, stored values are reloaded and compared with the ones used inside the loop using pairwise `xor` operations, the results of which are summed and compared with zero

in a single step. Such a block must be placed on every exit of the loop ensuring the detection of a variable corruption leading to another exit that does not depend on the corrupted variable. Hence, this new block is duplicated for each exit e^{out} before being inserted in the CFG by the updateIOpsCFGout function (cf. Algorithm 1).

In the case of an operand considered as loop-invariant for multiple loops and used in their associated slices, there is no need to store this operand multiple times. In such a case, the variable should not be stored for each loop, but rather only once in a block that dominates all the loops using it, such as in the block containing its definition if it exists or in the function entry, otherwise. The earlier the variable is stored, i.e., the closer to its definition, the shorter the attack surface to corrupt it. Hence, this step looks up where to place the store of loop-independent variables to minimize its attack surface.

Updating the CFG of the Entire Loop. During the processing step of each exit block e^{out} associated to an exiting block e^{in}, a new basic block bb_{out} has been created. When effectively exiting the loop, the processor should execute it before branching to e^{out}. The updateExitCondCFGout function makes the basic block bb_{out} to become the target of the exiting block e^{in} instead of e^{out} (Algorithm 1 line 10). When the original and duplicate exit conditions do not match, a fault must be detected and control flow links to an error handler. Thus, bb_{out} ends with a conditional branch on the duplicated condition and targets either e^{out} if the exit condition holds or the error handler *eh* if not.

Moreover, in the last step of Algorithm 1 (line 15), a basic block bb_{iops} is created in order to check loop-invariant operands. Such a block must be executed before branching to any exit block e^{out}. Hence, for every e^{out} (which can be common to several exiting blocks), updateIOpsCFGout inserts a copy of bb_{iops} such that it becomes the target of all previously inserted bb_{out} instead of e^{out} (Algorithm 1, line 17). This block has two successors: the original exit block e^{out} and an error handler block. When the loop-invariant variables pass the fault detection check, the execution proceeds normally with the e^{out} block; otherwise, it branches to *eh*.

Similarly to the in-loop CFG update, care must be taken in the placement of the additional blocks. To avoid inserting a new error handler after each checking block and, more importantly, to ensure a strict separation of the checking code with any other code: all bb_{out} blocks (created for one or several e^{in}) must be placed sequentially in any order and immediately before bb_{iops}. Also, bb_{iops} must be placed right before e^{out}. One may use the error handler block created for bb_{in} associated with the exiting block e^{in}. This last placement constraint ensures, in the case of a fault skipping a branch of one of these blocks, the execution cannot continue in a unpredictable part of the code with unpredictable effect. Thus, if a fault targets the branch instruction of bb_{iops}, the execution will fall into e^{out} without any impact on the loop iteration count due to the single fault assumption of the model. Similarly, if a fault targets the branch instruction of one bb_{out}, the execution will fall either in another bb_{out} block or in bb_{iops}. In Figure 1, showing a loop (left part) and its hardened version (right part), dashed arrows indicate fall-through blocks.

The placement requirement for multiple checking blocks bb_{out} associated with the same exit block is represented with dotted arrows, expressing their sequential placement immediately before bb_{iops}. Only one checking block will have bb_{iops} as fall-through.

Duplication Limitations. To protect against one fault, every instruction present in a slice has to be duplicated. However, some instructions cannot be duplicated when this would induce observable side-effects. For instance, it is not safe to duplicate a volatile load or a function call without any information on the callee. If a slice contains such an instruction, the associated exit condition cannot be protected. To let the user keep track of the partial or complete protection achieved on a given sensitive loop, the hardening pass emits a warning for every exiting block that cannot be handled. The user may then check that some of these loop exit points are not sensitive (loops often detect early conditions to abort the current operation, and these do not necessarily carry sensitive

information), or she may reorganize the code until the protection can be applied, or implement post-pass manual hardening as a last resort.

Outermost Loops First. In the presence of nested loops, the order in which loops are processed may impact the cumulative overhead due to instruction duplication and block creation. When two nested loops have a common exiting block (e.g., a `return` statement), the slice computed by the backward analysis of the inner loop is necessarily included in the one computed for the outer one. Some loop-independent variables for the inner loop may be loop-dependent for the outer loop, leading to a bigger slice. As a result, securing an exiting branch for the outer loop simultaneously hardens it for the inner loop. Hence, an optimization consists in securing loop nests from the outermost inwards, marking exiting blocks as secured before processing inner loops.

Special attention must be paid in the case of an exiting block ending with a switch instruction. The securing scheme must be adapted when one successor bb_s of the exiting block is both an in-loop successor for the outer-loop and an exit block for the inner-loop. Recall that when securing a switch, one single bb_{in} block is added for all in-loop successors whereas each exit block has its own checking blocks bb_{out} and bb_{iops}. bb_s must be hardened separately to implement its dual protection as an in-loop successor and an (inner loop) exit, adding checking blocks for both roles.

3.4 Security Analysis

This section discusses the robustness of the proposed hardening scheme. As presented in Section 2.2, we assume the attacker can corrupt the value of a register or skip an instruction, and that only one such fault may occur during the execution of a sensitive loop.

Let us first recall the design principles of the hardening scheme. The backward analysis computes the list of instructions and operands as well as conditional branches involved in an exit condition. Loop-dependent scalar operands and instructions are duplicated, inserting a slice in the data flow fully isolated from the original one. Conversely, loop-invariant operands are backed up in memory before the loop and are available to both data flows. Based on these duplicated or stored operands, faulty exit conditions are detected in additional checking blocks: before in-loop hardened branch successors to check the right path is taken, and before every exit block to ensure this exit is the expected one. An additional checking block reloads loop-invariant operands to verify their integrity. Faults impacting branches (corruptions of its conditions or branch skips) are captured via the sequencing of checking and error handler blocks placed on all possible following paths.

The security analysis must consider both instruction skip and register corruption (except the program counter), at any execution point of a loop, as specified in the fault model. However, skipping an instruction with a destination register is equivalent to corrupting this register if this instruction does not update the flags (branch condition codes). If this instruction updates the flags, there are two cases: if the flags are not alive, again the fault is equivalent to a single register corruption (the destination register); if the flags are alive, their simultaneous corruption will only affect the same branch condition or the computation of the same exit condition as the corrupted register (original or duplicate) and the second one (duplicate or original) will be correct. Thus, skipping an instruction that updates a general register and potentially the flags has the same effect as corrupting its destination register. The same reasoning applies to all instructions with multiple destination registers when all of these belong to the same slice, e.g., memory access with post/pre-increment/decrement. For all faults targeting such instructions, we will only consider register corruption for security analysis. Instruction skip must only be considered in case of a fault targeting a branch. Notice the case of stores to memory does not need to be considered since slices do not contain such instructions. Finally, the ISA may include instructions with multiple destination registers that do not necessarily belong to the same slice; an example of this rare case is discussed in Section 4.2.3.

The algorithm traverses use-def chains but not general read-after-write data dependences in memory. If a termination condition depends on data stored earlier in the loop, it is possible that the earlier store captured a corrupted value. Protection from such faults would involve a check before every store aliasing with a subsequent load—in the same or later iteration—in the backward slice of an exit condition. The algorithm could be extended to insert such checks, but it would be expensive and the benchmarks did not expose such data dependence chains in exit conditions.

Branch Instruction Skip. Skipping an in-loop duplicated branch will have no impact as execution will sequentially continue in the block originally targeted by the branch for the path chosen.

Skipping a branch at the end of a block added to secure a loop exit will fall into the following block in the layout. Such a block is either followed by an error handler (case bb_{in}), or it is outside the loop (case bb_{out} or bb_{iops}) and will either be detected or will have no impact on the loop iteration count.

Skipping an original internal branch instruction will go to one of the inserted checking blocks. If the execution path is altered by the fault, the duplicated condition will not match and the fault will then be detected. Otherwise, the fault will just be silent. And skipping a loop exit branch will force the execution flow to fall into the checking block bb_{in}. If the loop should stop iterating, the fault will also be detected; it will be silent, otherwise.

Unconditional branches can appear in the loop. If skipped, the execution flow continues in the fall-through block, which may either belong to the loop—and may alter some data—or be outside the loop—in case of a non-exiting latch block, for instance.[6] To secure an unconditional branch, it is sufficient to duplicate it. Since such duplicates may be eliminated as dead code, unconditional branches must be secured downstream from every pass able to eliminate dead code.

Register Corruption. A fault can target a register containing a loop-invariant operand involved in any condition of a hardened in-loop branch or any exit condition. In such a case, the original and duplicated conditions will be impacted but the fault will finally be detected by one of the bb_{iops} blocks in charge of controlling the correctness of all loop-invariant operands involved in at least one condition of a hardened in-loop branch or one exit condition. Indeed, such a block is placed on every exit of the loop.

Otherwise, if a fault targets an operand in the duplicated data flow of the computation of an exit condition, it will have no consequence on the original data flow and loop iteration count unless the fault is detected, either inside the loop in bb_{in} or outside in bb_{out}.

If a fault targets the original data flow of the computation of an exit condition, it can either reduce the number of iterations and be detected outside the loop or add an iteration and then be detected inside the loop by the added checking block.

Similarly, if a fault affects the original condition of an in-loop branch, altering the in-loop executed path, it will be detected by the non-faulted duplicated condition checked on the wrong executed path. If the computation of a duplicated branch condition is altered by register corruption, it will also be detected by the added checking block, even if the right path has been taken.

This reasoning covers all branches to be secured, following the inductive argument of the backward slice construction in Section 3.3. If a branch br impacts an exit condition, it means there are multiple paths leading to this exit condition. The SSA form guarantees the presence of a ϕ node between br and the exiting condition. This ϕ node will be part of the backward slice because the condition (transitively) depends on it.

Note that some faults may lead to raising an exception (e.g., division by zero) or make the loop infinite (e.g., setting the increment to 0). These last cases are not considered as a security breach as the system will be either interrupted or unresponsive.

[6] A block with a back-edge to the header.

4 LOOP HARDENING PASS AND COMPILATION FLOW

Source code hardening can be altered by an optimizing compiler, as explained in Section 1. This is also the case of compiler-assisted hardening, considering the downstream passes in the compiler. In the absence of programming language semantics and a compiler framework to systematically reason about the preservation of security properties and countermeasures, we resort to a pragmatic analysis of the compilation passes and their transformational power. This analysis, in turn, lets us select a relevant position for the loop hardening pass in the compilation flow, so that the countermeasure is preserved in the generated assembly code while retaining the ability to apply aggressive performance and code-size optimizations.

At IR level, data and control flow information are easily accessible. It is also interesting to schedule the pass early enough to preserve as much target-independence as possible, and also to retain the ability to apply cleanup passes. Now, operating at IR level is not a fundamental constraint but rather a choice aiming at providing a security-oriented, optimizing compilation flow.

To determine a safe and profitable position relatively to upstream and downstream analyses and transformations, we analyzed the typical passes in a generic compiler framework as well as a concrete incarnation of LLVM. This section presents the result of this analysis.

4.1 Upstream Passes and Placement in the Compilation Flow

Let us study the passes *providing essential information to apply the algorithm*, *the interfering ones that need to be scheduled before the hardening pass*, and *the potentially beneficial passes that improve code quality after loop hardening*. The positioning of loop hardening in LLVM is discussed accordingly.

4.1.1 Upstream Passes. The *register promotion* pass attempts to replace memory accesses with scalar variables, hoping these will be eventually allocated to registers. It is useful to eliminate spurious loads, occasionally exposing data dependences into explicit def-use chains. This is very convenient to improve our ability to construct backward slices that remain free of load-store aliasing. Such a *mandatory pass* has to be scheduled upstream.

Redundancy elimination passes such as *global value numbering* and *common sub-expression elimination* aim at simplifying computations and at replacing computations with temporary variables. Such passes can severely alter or even remove the code inserted by our hardening pass. *Induction variable simplification* tries to identify a canonical induction cycle from which other induction variables can be derived; it may radically transform the conditional expressions in exit conditions. *Loop strength reduction* transforms expensive operations into iterations of less costly ones. It may also radically change conditional expressions or create additional derived induction variables. All these *interfering passes* need to be scheduled upstream to avoid compromising the protection.

Loop invariant code motion hoists and sinks loop invariant instructions out of the loop. It reduces the size of the loop body and removes redundant computations that may otherwise be swallowed into backward slices. The effects are positive on the size and overhead of countermeasure. Several compilers (including GCC and LLVM) further normalize the SSA form by enforcing a property known as *loop-closed SSA*. Such compilers implement passes inserting a ϕ node at loop exits for each variable defined inside the loop and used outside the loop. This simplifies the application of program transformations within the scope of a given loop by isolating them from follow-up control flow. Again, it is profitable to schedule such *beneficial* passes before loop hardening.

4.1.2 Pass Placement in LLVM. In a typical incarnation of the LLVM compiler, among the three aforementioned classes, *loop strength reduction* (LSR) is the last scheduled pass in the flow. As it implements a target-dependent optimization, it is scheduled in the backend but still at IR level,

shortly before machine code generation. Then our hardening pass also belongs to the backend, positioned immediately after LSR and before machine code generation. It benefits from many normalizing and simplification passes, and much fewer optimizations have a chance of interfering with the inserted countermeasure. Also, while the pass is formally scheduled in the backend, it still operates at IR level and remains target-independent. We may thus support multiple targets with a very reasonable effort.

4.2 Downstream Passes and Backend/Target-Dependent Adjustments

To validate this choice of pass ordering, it is necessary to study the analyses and transformational power of the downstream passes in a typical backend compiler. While no fully generic investigation is possible, we can reason on the typical lowering and optimization passes found after SSA destruction in a low-level IR, and verify our assumptions on concrete backends. We used the ARM backend for this purpose, simulating fault attacks targeting instructions and registers of sensitive loops (see Section 5.6). In most cases, we were able to check that potentially harmful code motion, redundancy elimination, or instruction selection on low-level code did not interfere with with the protected code.[7] Let us review the specific needs for adjustments to avoid interference.

4.2.1 Basic Block Layout. As we have seen in the previous section, the hardening algorithm relies on specific checking and error handling blocks to be laid out as fall-through control flow. A particular layout can be enforced by means of a partial ordering constraint on the CFG [24].

Independently from loop exiting, some loop blocks end with an unconditional branch that may not even be visible at IR level (it depends on the basic block layout). Skipping this unconditional branch can exit the loop without any redundant check or affect the in-loop control flow. Without further information on basic block layout, this may have unpredictable effect on the loop exit and iteration count. Every in-loop unconditional branch should then be duplicated to avoid this issue.

Considering the ARM backend in LLVM, two transformations can modify the basic block layout, control flow optimizer, and branch folding. The branch folding pass has been modified to avoid movement of blocks tagged as checking blocks such that they are kept as fall-through of their predecessor without preventing any other folding optimization from being applied. On the contrary, the control flow optimizer pass has been entirely disabled and the potential overhead associated is included in every result in the next section. A short dedicated pass to duplicate unconditional branches of the loop has been added at the end of the backend, before code emission.

4.2.2 Spills. In a typical register allocator, a variable that is considered read-only during a sub-interval of its live range will not be stored back at the end of this sub-interval in case of spills. Any corruption on the register holding this variable may not be detected when reloading the spilled value, while the faulty one may have compromised downstream computations in the slice. Interestingly, this issue may only arise in the case of a loop-invariant variable, since other variables occurring in a slice are systematically duplicated. To solve this issue, an additional store should be forced at the end of any sub-interval of a variable's live range, as soon as there exists a path from this sub-interval to a reload of the variable. This modification has not been implemented yet.

4.2.3 Local Optimizations. Some constants at IR level may be promoted to a register through one or several instructions depending on the size of the constant and addressing mode constraints (e.g., the bit width of immediate operands). This promotion does not compromise the protection per se, yet the constants appearing in the original instruction and its duplicate may be recognized as identical and loaded into a single register, exposing weakness against register corruption.

[7] In LLVM, this includes machine loop-invariant code motion and machine common sub-expression elimination.

The optimization that performs such *identical constant elimination* in secured loops should be deactivated. In LLVM, this is directly implemented during instruction selection and has not been modified yet.

Similarly to identical constants, the backend can detect *duplicate instructions* with identical operands, and eliminate one of them. This is most typical of redundant loads. This issue can be solved either at backend level by disabling the elimination of redundant instructions, or at IR level by qualifying the duplicated load as volatile. We chose the latter in our LLVM implementation.

The folding of two IR instructions into a single assembly instruction can occur in more scenarios. As an example, multiple consecutive memory access can be loaded as a single instruction (often called "load pair"). A single skip on such an instruction results in multiple faults at IR level. This is permitted by the fault model but is not covered by our loop hardening scheme. It is sufficient to deactivate such *instruction folding*. Notice, this deactivation can be slightly refined if it would be possible to identify, after the hardening pass, which IR instructions come from an original or duplicate slice. One could then deactivate instruction folding for pairs of instruction belonging to the slice and its associated duplicate. Indeed, following the security analysis of flags in Section 3.4, two instructions folded in a single one are dangerous only if one instruction belongs to the original slice and one to the duplicated slice.

5 IMPLEMENTATION AND EXPERIMENTAL RESULTS

Let us now report on the implementation of the loop hardening pass.

5.1 Implementation

We implemented our algorithm as an IR pass of LLVM [29] version 4.0.0. We modified the ARM back-end to add the securing pass and adapted it following the lines of the previous section. The x86_64 back-end has only been modified to include the pass for functional validation purposes. As shown in this section, we validated the pass with three targets: x86_64, ARMv7m/Thumb2, and ARMv7a ISA.

Loop Selection. Since the sensitive loops may not be the majority in general, it is important to let the developer decide which loops should be hardened. We offer this degree of control by means of source code attributes with three hardening options: the default option activates loop hardening for every loop in the compilation unit; the function attribute *__attribute__((secLoops))* limits it to loops in the annotated function; and finally, an intrinsic function call may be inserted immediately before a loop control structure to mark it for hardening.

Limitations. Some loops cannot be secured due to the presence of instructions that can not be duplicated. We implemented a white list to determine if all instructions of a given slice can be duplicated. The current implementation allows every arithmetic, bitwise, and cast operator, as well as SSA ϕ nodes and loads. All other instructions are considered unsafe to duplicate. As discussed earlier, when an instruction of a slice is not in the white list, a warning is emitted to inform the user that the protection could not be applied to a particular exit condition.

Another limitation concerns the presence of an inline assembly piece of code inside the loop body. In this case also, a warning is emitted as the securing scheme cannot be safely applied.

Overall, our approach errs on the side of safety, signaling to the user any loop that does not match the semantical requirements of our hardening algorithm and implementation.

5.2 Experimental Setup

The goal is to evaluate the automatic, end-to-end hardening and robustness on fault attacks. To highlight the generality of the hardening scheme and establish an upper bound on the expected

overhead, the following experiments force the hardening of all loops, independently of any user code annotation. In a deployed scenario, only specific loops are sensitive and should be hardened.

For this purpose, we consider three well-understood production versions of cryptographic protocols: symmetric cryptography—`aes`—public-key cryptography—`ecc`—and hashing—`sha`. These codes contain sensitive loops that can leak information if their iteration counts are corrupted and represent typical real targeted code to be hardened.

In addition, we characterize the hardening pass's coverage of a wider range of loops in both security-oriented and generic benchmarks, testing its functional correctness, and analyzing its size and performance overhead. We selected 15 additional benchmarks: `pgp` and `blowfish` from the MiBench 1.0 suite [22]; `gzip 1.2.4`[8] compression application for its high density of loops over a relatively small source code; three larger security-sensitive benchmarks `openssl 1.1.0c`, `gmp 6.1.2`, and `sqlite 3.15.1`[9]; and all the SPEC CPU2006 integer benchmarks implemented in C [23]. They have been chosen to stress our hardening pass in a wider variety of control and data flow scenarios, including large loops with complex control-flow.

We conduct the full security evaluation on two ARM targets: an NXP board, LPCXpresso 1343 rev. A, with a low-end 32-bit Cortex-M3 from the ARM Cortex-M family implementing the ARMv7m/Thumb2 ISA, representative of deeply embedded devices; and a higher-end Raspberry Pi-3 board with an ARM Cortex-A53 implementing the ARMv7a ISA and running the Raspbian GNU/Linux operating system.

On the NXP LPCXpresso board, the executable code is loaded from a host PC through the board's JTAG port. The contents of memory and general-purpose registers is dumped at the end of the execution. Due to limited memory and I/O capability, several benchmarks could not be run on this LPCXpresso board.

To further emphasize portability, we also replicated the functional and performance experiments on an x86_64 target: an Intel Core-i3 5010 processor running the Debian 8 GNU/Linux operating system.

Experiments cover all standard optimization levels (`-O1`, `-O2`, `-O3`, `-Os`, `-Oz`)[10] unless specified otherwise.

5.3 Loop Coverage

In order to quantify our algorithm's ability to handle a large variety of loops, we analyze, for each benchmark, the percentage of loops that can be hardened automatically: the fully secured—all exits are secured; the partially secured—at least one exiting is secured; –and the not secured ones. Results are given in Figure 3.

Considering all benchmarks, the percentage of fully hardened loops varies from 45% to 100%, and it never goes below 70% when considering the criterion that at least one loop exit has been successfully hardened.

We investigate the characteristics of unprotected termination conditions. Our pass fails to harden a specific loop when a backward slice contains a non-duplicable instruction. For example, this is the case of up to half of the computed slices for `openssl`, `sqlite`, and `403.gcc`. About 80% of the non-duplicable instructions are function calls and 20% are volatile memory accesses. The reader should note that volatile memory accesses can not be handled by any software protection based on duplication [6, 32, 45].

[8] http://www.gzip.org.
[9] http://www.openssl.org, http://gmplib.org, and http://www.sqlite.org.
[10] LLVM strongest code size optimization level.

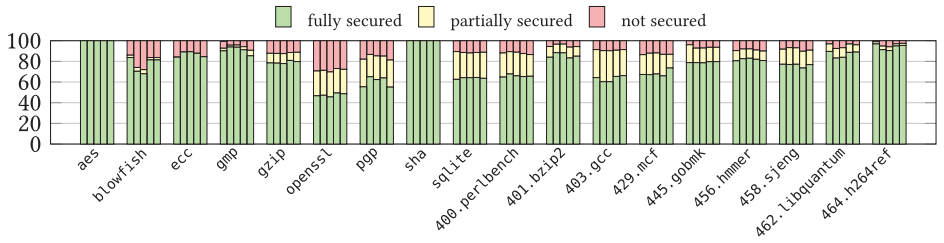

Fig. 3. Loop coverage of the hardening scheme at IR level—ordered by compiler option -O1, -O2, -O3, -Os, -Oz.

On the contrary, leveraging semantical information about functions (e.g., *pure* or *const* attributes), it would be possible to harden exiting blocks whose backward slices contain only side-effect-free calls.

In any such cases, the compiler emits a warning to let the user localize the problematic exit condition and attempt to adapt the implementation, annotating, inlining, or hoisting function calls accordingly.

Overall, this validates our design of a generic algorithm capable of handling complex control flow and conditional expressions.

5.4 Functional Evaluation

Let us now validate the functional correctness of the hardened benchmarks. The experimental protocol differs slightly across benchmarks.

We validated the three cryptographic benchmarks `aes`, `ecc`, and `sha` using a test procedure covering the different options of the library, where the output produced for a given input is compared to the expected, pre-computed one.

For `gzip` and benchmarks from the MiBench suite—`pgp` and `blowfish`—we compressed and decompressed, and encrypted and decrypted, respectively, a 420MB file. The resulting file is compared to the initial one for functional testing. We used the auto-test feature of `openssl` and `gmp` to check the compiled library and the full validation infrastructure of SPEC CPU2006 benchmarks. Finally, for `sqlite`, we reproduced the *speed* test available from the official website.[11] These different evaluation methods have been used considering all optimization levels.

All functional tests succeeded, highlighting the correct execution of the selected benchmarks after automatically hardening their loops.

5.5 Performance and Overhead

Regarding runtime performance or code size overhead, as the `aes`, `ecc`, and `sha` implementations are dedicated to ARM targets, experiments are only performed on both the ARM boards. For all others benchmarks, experiments have been realized on Cortex-A53 and x86.

Performance. Performance overhead measurements are based on 10 runs of the benchmarks presented in Section 5.4; Figure 4 presents the average, excluding extreme values.

Obviously, the performance impact depends on the time spent in loops and is higher for loops with short bodies. For example, on the ARM target, hardening the `sha` benchmark composed of numerous simple loops comes with high overhead, whereas `ecc`, which is composed of few but large loops, incurs very low overhead. For all benchmarks, overhead ranges from 0 to 50% with an

[11] https://sqlite.org/speed.html.

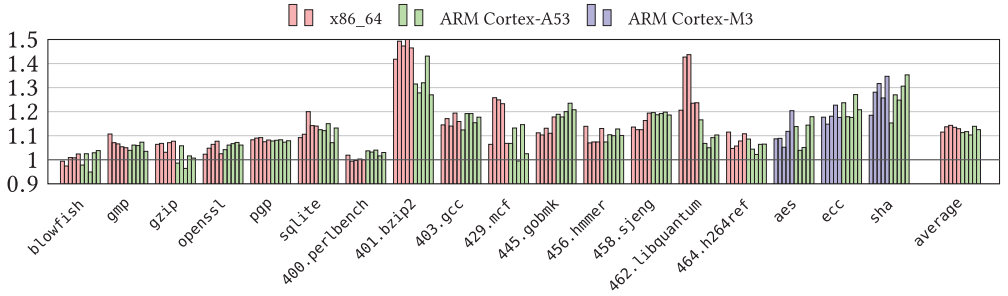

Fig. 4. Performance overhead ratio—ordered by compiler option -O1, -O2, -O3, -Os, -Oz.

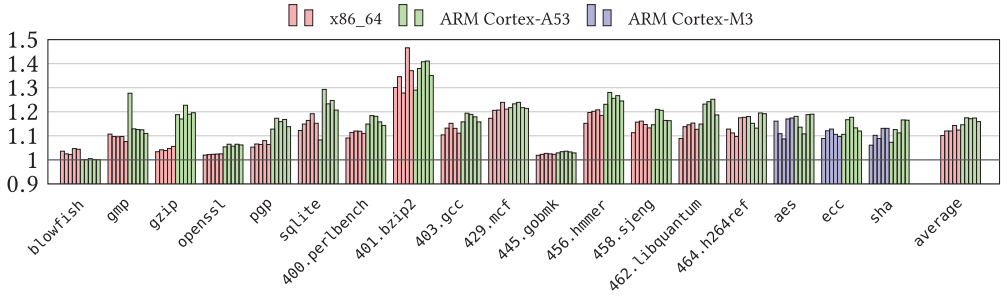

Fig. 5. Code size overhead ratio—ordered by compiler option -O1, -O2, -O3, -Os, -Oz.

Table 1. Compilation Time (Percentage of Total Compilation Time)

Compilation time (%)	aes	blowfish	ecc	gmp	gzip	openssl	pgp	sha	sqlite	400.perlbench	401.bzip2	403.gcc	429.mcf	445.gobmk	456.hmmer	458.sjeng	462.libquantum	464.h264ref
Canonicalize natural loops	0.8	0.4	1.2	0.6	0.9	0.6	0.7	0.6	0.3	0.6	0.9	0.5	0.9	0.7	0.7	0.8	0.9	0.8
Loop-invariant code motion	1.8	2.5	2.1	1.8	4.0	1.8	3.4	1.7	3.7	1.7	4.0	2.4	4.1	2.8	3.1	3.1	2.1	3.9
Loop strength reduction	1.7	2.1	3.0	13.0	3.2	1.3	2.9	4.9	0.6	1.6	2.4	0.9	4.4	2.3	4.0	3.4	2.7	5.3
Loop hardening	**0.1**	**0.1**	**0.1**	**0.6**	**0.2**	**0.2**	**0.2**	**0.1**	**8.4**	**1.0**	**0.8**	**1.8**	**0.5**	**0.3**	**0.5**	**0.4**	**0.4**	**0.2**

average of 12.5%. Thus, the performance overhead of the proposed hardening is very small, and much lower than the overhead induced by generic instruction hardening schemes: the latter, often based on systematic instruction duplication, induced a typical overhead of more than 100% [6, 32, 45]. Such code hardening applied to sensitive parts allows to limit the overhead (e.g., to 40% for an AES cipher protected against instruction skip only [32]), but is still higher.

Code Size. Figure 5 presents the ratio between the size of hardened benchmarks compared to their original versions. These are computed from the cumulated sizes of all (statically linked) object files included in the benchmark.

Code size overhead varies from less than 1% to 43%. It mainly depends on the fraction of loop code and on the loop properties. For example, `401.bzip2` is composed of many complex, nested loops inducing higher code size overhead than for other benchmarks. The average overhead is only 14%, strongly supporting the proposed countermeasure dedicated to guaranteeing the loop iteration count and its termination through the correct path.

Compilation Time. Table 1 presents the percentage of compilation time spent in the loop hardening pass as well as in other (loop) optimization passes when compiling at optimization level -O2. For all benchmarks but `sqlite`, the hardening pass takes a very low percentage of the

compilation time. For `sqlite`, as source code is fully contained in a single file (7MB), it offers more opportunities for optimization in the upstream compilation flow, e.g., through function inlining. As a result, the IR is composed of large loops with very long switch statements enclosed in loops with numerous exitings, requiring more time to harden.

5.6 Security Evaluation

We designed a fault injection simulator based on gdb to assess the security of the hardened applications. The simulator relies on gdb's python scripting capability. The principle is to execute in debugging mode an application up to a breakpoint set at the instruction where one chooses a fault to be simulated. When execution stops, a transient fault is injected, simulating an instruction skip or register corruption as described in the fault model (cf. Section 2.2), then the execution proceeds.

To analyze a faulted run, the expected number of iterations of each secured loop must be known. Indeed, it is not sufficient to compare normal and faulty outputs: a fault may target an instruction not related to the loop iteration count and exit, hence, will alter the output without compromising the security property targeted by our securing scheme. To retrieve expected and executed iteration counts, the hardening pass has been complemented with the ability to instrument the generated assembly code with an additional global variable for every secured loop. This variable is incremented in the loop header. We also use accesses to this global variable to identify loop boundaries in the binary code. The expected iteration count of each secured loop is then determined by pre-executing the application without fault injection. Finally, the security evaluation consists in comparing this value to the one obtained when a fault is injected at a specific point and iteration of the loop body.

The fault simulator targets the ARM processors. For simulation time reason, we only conducted the security evaluation on benchmarks runnable on both these targets and that represent real targeted code to be hardened—`aes`, `sha`, and `ecc`. We evaluated all loops that can be automatically hardened, on both their original and secured versions.

To cover the space of register corruption effects, we ran a fault simulation for every general purpose register (`r0` to `r12` and `lr`) and every instruction of the loop, considering the three corruption models introduced in Section 2.2: stuck at `0`, stuck at `0xFFFFFFFF`, and random corruption. For instruction skips, we ran a fault simulation for every instruction of the loop. As faults are transient, we considered three injection times: the first iteration, an iteration midway through the loop execution, and the last iteration.

Table 2 presents the fault simulation results for both the initial and the hardened code compiled at `-O2` and `-Oz` optimization levels. Faults may have four different effects: no effect (e.g., corrupting a dead register), lead to a hard fault (e.g., load from invalid address), may be detected (error handler call), or alter the iteration count without being detected.

The entire set of benchmarks consisted in a total of around 615,000 fault injections on the original versions and around 913,000 fault injections on the secured ones, spread over 115 different loops (including loop nests); this difference is a consequence of the increased attack surface of the hardened loops. While 29,083 faults led to a successful attack on the original, unprotected versions, only 454 faults went undetected on the secure versions, reducing the attack success rate by 98% (and more than 99% when considering instruction skip only).

The undetected faults come from three sources, all associated with interferences from downstream compilation passes. Solutions to prevent these interferences have been discussed in the previous section, but we did not yet implement them. The most frequent source is that the address of a global variable is not available at IR level; hence, it can not be duplicated when loading the value of the variable. If the register containing its address is corrupted before the effective load, both original and duplicate load instructions, which use the corrupted register, will be affected

Table 2. Fault Simulation for Security Evaluation

				Instruction Skip					Register Corruption				
				No Effect	Hard Fault	Detected	Undetected	Total	No Effect	Hard Fault	Detected	Undetected	Total
Normal version	M3	aes	O2	399	79	0	**55**	533	15956	5519	0	**911**	22386
			Oz	675	108	0	**92**	875	27870	7253	0	**1627**	36750
		ecc	O2	798	104	0	**115**	1017	33052	8467	0	**1195**	42714
			Oz	1884	233	0	**223**	2340	38861	8658	0	**1621**	49140
		sha	O2	1245	113	0	**366**	1724	57778	8210	0	**6420**	72408
			Oz	1194	101	0	**279**	1574	51809	11926	0	**4053**	67788
	A53	aes	O2	434	57	0	**58**	549	15796	5005	0	**610**	21411
			Oz	538	62	0	**65**	665	19623	5425	0	**887**	25935
		ecc	O2	1641	230	0	**259**	2130	59959	20721	0	**2390**	83070
			Oz	461	136	0	**129**	726	18830	8510	0	**974**	28314
		sha	O2	1185	90	0	**341**	1616	48159	12576	0	**2219**	62954
			Oz	1514	64	0	**618**	2196	67318	14750	0	**3576**	85644
	Total			**11968**	**1377**	**0**	**2600**	**15945**	**455011**	**117020**	**0**	**26483**	**598514**
Hardened version	M3	aes	O2	533	93	100	**1**	727	22204	6386	2419	**29**	31038
			Oz	1641	235	249	**0**	2125	69040	13424	6435	**14**	88913
		ecc	O2	1028	106	302	**8**	1444	44694	8680	7020	**45**	60439
			Oz	1251	66	279	**1**	1597	51982	7375	5560	**47**	64964
		sha	O2	1625	202	489	**0**	2316	64268	16178	11063	**160**	91669
			Oz	1424	208	264	**0**	1896	57462	13468	5200	**55**	76185
	A53	aes	O2	583	102	122	**2**	809	18920	4607	2289	**12**	25828
			Oz	1373	90	225	**0**	1688	52028	8476	5314	**30**	65848
		ecc	O2	2105	284	539	**4**	2932	79695	23884	10662	**23**	114264
			Oz	1353	129	319	**1**	1802	51191	12728	6346	**22**	70287
		sha	O2	1661	106	560	**0**	2327	62744	17347	10631	**0**	90722
			Oz	2210	145	484	**0**	2839	84508	17954	8040	**0**	110502
	Total			**16787**	**1766**	**3932**	**17**	**22502**	**658736**	**150507**	**80979**	**437**	**890659**

and the fault will not be detected. The second source of undetected fault concerns large constant operands that cannot be used as an immediate field in ARM assembly, and has, thus, been promoted to a register by the backend. This register has then no duplicate. Both cases have been discussed in Section 4.2.3 and we proposed modifications of downstream passes to eliminate them. The last case of undetected fault is due to the spill of a loop-invariant operand. This causes an uncorrupted value to be loaded in the bb_{iops} check block at loop exit, whereas a corrupted value has been used in the loop computation. This case and its solution have been described in Section 4.2.2.

These experiments confirm that our loop hardening scheme and its LLVM implementation are highly effective at securing sensitive loops, detecting 98% of harmful attacks. Moreover, this detection rate could be raised to 100% with further modifications of the backend compiler.

6 RELATED WORK

Let us now discuss the most closely related work on automatic and low-level code hardening. Our countermeasure should be understood as one additional blade in a Swiss army knife of protections against fault attacks, under specific fault models, and within a code size and performance budget.

To withstand a single bit-flip due to hardware transient faults, Oh et al. [39] proposed a software redundancy approach wherein all instructions are duplicated and appropriate validation instructions are inserted. More recently, Barenghi et al. proposed a duplication (and triplication) scheme in order to thwart fault attacks on an implementation of AES [5]. Their technique detects instruction skips and some instruction replacements but does not fully capture register corruption. Moro et al. proposed a formally-verified duplication scheme to protect low-level code against instruction skip only [32]. It has recently been automated by Barry et al. in LLVM [6]. These instruction-level protections may be applied to loop bodies, but they do not capture the iteration count and may thus fail to detect faults affecting termination conditions. It appears necessary to extend such techniques for a more comprehensive coverage of loop security properties, as we propose in this article.

De Keulenaer et al. implemented a loop hardening technique in Diablo, a link-time rewriting tool [14]. The scope of the protection scheme is quite limited due to restrictive prerequisites. Candidate loops must contain a loop counter recoverable through the analysis of the binary code and

updated exactly once per iteration (a simple induction variable). Moreover, exit conditions may only compare this counter with a loop invariant. Our benchmarks show that the number of loops that fit these requirements is low, particularly after the application of compiler optimizations.

Control-flow integrity techniques are designed to ensure the validity of jump targets [1]. In the presence of fault attacks, fine-grain integrity is requested to detect a possible instruction replacement [13, 28, 48]. These techniques rely on the control-flow signature of each basic block [38] and on a complementary software or hardware mechanism to validate branch targets [13, 48]. Theses approaches cannot detect loop counter disruptions due to register corruption as the flow of instructions is not affected.

In their work [45], Reis et al. implemented a software-only fault detection system. It is based on the duplication of the entire program where the original and the copy use different registers and memory locations. It also follows a signature-based control-flow checking scheme. While duplicating all the code, their method also detects instruction skip and register corruption as defined in our fault model. This full duplication would effectively secure loops, but it may not be compatible with embedded constraints. Indeed, it was originally targeted to high-performance platforms with large memory. Reducing this overhead would involve focusing the duplication to specific operands and branches, which is precisely the core of our new technique. Also, it operates at a low level for a specific instruction set, and adapting our algorithm to such a low level code raises the need to recover loop and precise data flow information.

A recent trend to protect sensitive applications from physical attacks is to transform the compiler into a hardening tool automatically, implementing protections against side-channel attacks such as code masking and random precharging [2, 8, 33]. These approaches cannot be directly compared to the scenario considered in this article, but they highlight the growing interest from the security community in applying countermeasures at compilation time.

7 CONCLUSION

We addressed the modeling and enforcement of a loop-centric security property: the fact that a given sensitive loop performs the expected number of iterations and takes the correct exit while being exposed to physical attacks. Our approach takes the form of an algorithm and compilation pass to automate the application of a software countermeasure. It offers protection against any single-event register corruption or instruction skip fault impacting the iteration count or termination of a sensitive loop. The application of the countermeasure takes the form of an SSA-based analysis and transformation, duplicating instructions and variables involved in the exit conditions of sensitive loops. We also analyzed the method's interaction with upstream and downstream compilation passes and proposed solutions to the unavoidable interferences with backend optimizations. We provided guarantees about the enforcement of the security property, empirical evidence about the effectiveness of its automatic application, robustness, performance, and code-size overhead. Future work includes the combination of multiple protections automated in a compiler, as well as experiments with a physical attack bench on real-world devices.

APPENDIX

We provide the generated ARM assembly code with and without loop hardening for two examples.

The first example given in Listing 4 performs a constant-time buffer comparison. The normal and hardened assembly codes are given in Listings 6 and 7. Blue-colored code represents instructions belonging to the computed (and duplicated) slice for the exit condition, and red-colored code deals with the loop invariant operands. The second source code example is given in Listing 5 and was also presented and discussed in Section 2.3. The loop body contains control flow that impacts the loop exit (orange-colored jump in Listing 8). This example illustrates how the loop securing scheme

```
int compare(int *A, int *B, int size) {
      int k, diff;
      for (k=0; k<size; k++)
         /* Constant time - no early exit */
            if (A[k] != B[k])
                  diff++;
      return diff;
}
```

Listing 4. Buffer comparison

```
int i = n;
while (i > 0) {
  if (i == 5) {
    i -= 2; /* ... */
  } else {
    i--; /* ... */
  }
}
```

Listing 5. Conditional induction

```
.entry1:
  push {r4, r5, r7, lr}

  mov lr, r0
  movs r0, #0
  cmp r2, #1

  blt .end
  movs r3, #0

.for.body:
  ldr.w r4, [r1, r3, lsl #2]
  ldr.w r5, [lr, r3, lsl #2]
  adds r3, #1

  cmp r5, r4
  it ne
  addne r0, #1
  cmp r3, r2
  blt .for.body

.end:

  pop {r4, r5, r7, pc}
```

Listing 6.

```
.entry1:
  push {r4, r5, r6, lr}
  sub sp, #8
  mov lr, r0
  movs r0, #0
  cmp r2, #1
  str r2, [sp, #4]
  blt .end
  movs r4, #0
  movs r3, #0
.for.body:
  ldr.w r5, [r1, r4, lsl #2]
  ldr.w r6, [lr, r4, lsl #2]
  adds r4, #1
  adds r3, #1
  cmp r6, r5
  it ne
  addne r0, #1
  cmp r4, r2
  bge .exit
  cmp r3, r2          /* bb_in */
  blt .for.body
.errorHandler:
  bl ErrorHandler
.exit:
  cmp r3, r2          /* bb_out */
  blt .errorHandler
  ldr r6, [sp, #4]    /* bb_iops */
  teq.w r2, r6
  bne .errorHandler
.end:
  add sp, #8
  pop {r4, r5, r6, pc}
```

Listing 7.

```
.entry2:
  push {r4, lr}

  mov r4, r0
  cmp r4, #1
  it lt
  poplt {r4, pc}

.while.body:
  cmp r4, #5
  bne .if.else

.if.then:
  /* ...then... */
  movs r4, #3

  b .while.cond

.if.else:
  /* ...else... */
  subs r4, #1

.while.cond:
  cmp r4, #0
  bgt .while.body

.end

  pop {r4, pc}
```

Listing 8.

```
.entry2:
  push {r4, r5, r6, lr}
  sub sp, #8
  mov r4, r0
  cmp r4, #1
  str r4, [sp, #4]
  blt .end
  mov r5, r4
  mov r6, r4
.while.body:
  cmp r5, #5
  bne .bb.else
 .bb.then:
  cmp r6, #5
  bne .error
.if.then:
  /* ...then... */
  movs r5, #3
  movs r6, #3
  b .while.cond
  b .while.cond
 .bb.else:
  cmp r6, #5
  bne .if.else
.error:
  bl errorHandler
.if.else:
  /* ...else... */
  subs r6, #1
  subs r5, #1
.while.cond:
  cmp r5, #1
  blt .bbOutside
  cmp r6, #0          /* bb_in */
  bgt .while.body
  b .error
  b .error
 .bbOutside:
  cmp r6, #0          /* bb_out */
  bgt .error
  ldr r6, [sp, #4]    /* bb_iops */
  teq.w r4, r6
  bne .error
.end:
  add sp, #8
  pop {r4, r5, r6, pc}
```

Listing 9.

Fig. 6. Source, original, and hardened assembly code for a buffer comparison (Listings 4, 6, and 7) and for a loop with internal control flow impacting the loop exit (Listings 5, 8, and 9).

deals with such cases. In the assembly secured version, in Listing 9, dark-red-colored code corresponds to basic blocks dealing with the loop exit condition and loop independent variable, whereas orange-colored code corresponds to the one related to securing the loop internal control flow.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*. 340–353.

[2] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi. 2013. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[3] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. 2011. An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 105–114.

[4] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. 2006. The sorcerer's apprentice guide to fault attacks. *Proc. of the IEEE* 94, 2 (Feb. 2006), 370–382.

[5] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. 2010. Countermeasures against fault attacks on software implemented AES. In *5th Workshop on Embedded Systems Security (WESS'10)*. ACM, Article 7, 7:1–7:10 pages.

[6] Thierno Barry, Damien Couroussé, and Bruno Robisson. 2016. Compilation of a countermeasure against instruction-skip fault attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems (CS2'16)*. 1–6.

[7] Gilles Barthe, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Jean-Christophe Zapalowicz. 2014. Synthesis of fault attacks on cryptographic implementations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*. ACM, New York, NY, 1016–1027. DOI: http://dx.doi.org/10.1145/2660267.2660304

[8] Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, Francois-Xavier Standaert, and Paolo Ienne. 2015. Automatic application of power analysis countermeasures. *IEEE Trans. Comp.* 64, 2 (2015), 329–341.

[9] S. Bhasin, P. Maistri, and F. Regazzoni. 2014. Malicious wave: A survey on actively tampering using electromagnetic glitch. In *International Symposium on Electromagnetic Compatibility*. 318–321.

[10] I. Biehl, B. Meyer, and V. Müller. 2000. Differential fault attacks on elliptic curve cryptosystems. In *Advances in Cryptology (CRYPTO 2000) (LNCS)*, M. Bellare (Ed.), Vol. 1880. Springer, 131–146.

[11] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 2001. On the importance of eliminating errors in cryptographic computations. *J. Cryptology* 14 (2001), 101–119.

[12] Common Criteria. Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 5.

[13] Jean-Luc Danger, Sylvain Guilley, Thibault Porteboeuf, Florian Praden, and Michal Timbert. 2014. HCODE: Hardware-enhanced real-time CFI. In *PPREW@ACSAC*. 6:1–6:11.

[14] R. De Keulenaer, J. Maebe, K. De Bosschere, and B. De Sutter. 2015. Link-time smart card code hardening. *International Journal of Information Security* (2015), 1–20.

[15] A. Dehbaoui, A.-P. Mirbaha, N. Moro, J.-M. Dutertre, and A. Tria. 2013. Electromagnetic glitch on the AES round counter. In *COSADE*.

[16] Hüseyin Demirci and Ali Aydın Selçuk. 2008. A meet-in-the-middle attack on 8-round AES. In *Fast Software Encryption: 15th International Workshop, FSE*. 116–126.

[17] Emmanuelle Dottax, Christophe Giraud, Matthieu Rivain, and Yannick Sierra. 2009. On second-order fault analysis resistance for CRT-RSA implementations. In *Third IFIP WG 11.2 International Workshop on Information Security Theory and Practice*. 68–83. DOI: http://dx.doi.org/10.1007/978-3-642-03944-7_6

[18] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens. 2016. FISSC: A fault injection and simulation secure collection. In *SAFECOMP*. 3–11.

[19] P. Dusart, G. Letourneux, and O. Vivolo. 2003. Differential fault analysis on AES. In *Applied Cryptography and Network Security (ACNS'03) (LNCS)*, M. Yung, Y. Han, and J. Zhou (Eds.), Vol. 2846. Springer, 293–306.

[20] Nadia El Mrabet. 2009. *What About Vulnerability to a Fault Attack of the Miller's Algorithm During an Identity Based Protocol?* Springer, Berlin, 122–134. DOI: http://dx.doi.org/10.1007/978-3-642-02617-1_13

[21] H. Eldib and C. Wang. 2014. Synthesis of masking countermeasures against side channel attacks. In *26th International Conference on Computer Aided Verification*. 114–130.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE (WWC'01)*. 3–14.

[23] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. DOI: http://dx.doi.org/10.1145/1186736.1186737

[24] G. Holloway and M. D. Smith. 2000. *An Extender's Guide to the Optimization Programming Interface and Target Descriptions. The Machine-SUIF documentation set.* Technical Report. Harvard University.
[25] Dusko Karaklajic, Jorn-Marc Schmidt, and Ingrid Verbauwhede. 2013. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 12 (2013), 2295–2306.
[26] C. H. Kim and J.-J. Quisquater. 2007. How can we overcome both side channel analysis and fault attacks on RSA-CRT? In *Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. 21–29.
[27] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Advances in Cryptology — CRYPTO'99*. LNCS, Vol. 1666. Springer, 388–397.
[28] J.-F. Lalande, K. Heydemann, and P. Berthom. 2014. Software countermeasures for control flow integrity of smart card C codes. In *Computer Security - ESORICS*. LNCS, Vol. 8713. Springer, 200–218.
[29] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*. 75–86.
[30] Nicolas Moro. 2014. *Sécurisation De Programmes Assembleur Face Aux Attaques Visant Les Processeurs Embarqués*. Ph.D. Dissertation. UPMC, Paris, France.
[31] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. 2013. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 77–88.
[32] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. 2014. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering* 4, 3 (2014), 145–156.
[33] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2012. Compiler assisted masking. In *Cryptographic Hardware and Embedded Systems – CHES 2012*. LNCS, Vol. 7428. Springer, 58–75.
[34] Nadia El Mrabet. 2013. Side Channel Attacks against Pairing over Theta Functions. *Cryptology ePrint Archive, Report 2013/386*. (2013). http://eprint.iacr.org/2013/386.
[35] S. S. Muchnick. 1997. *Advanced Compiler Design & Implementation*. Morgan Kaufmann.
[36] Frédéric Muller. 2003. *A New Attack against Khazad*. Springer, Berlin, 347–358.
[37] Shoei Nashimoto, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. 2016. Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering* (2016). DOI: http://dx.doi.org/10.1007/s13389-016-0136-3
[38] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Trans. Reliability* 1, 51 (2002), 111–122.
[39] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. 2002. Error detection by duplicated instructions in superscalar processors. *IEEE Trans. Reliability* 1, 51 (2002), 63–75.
[40] S. Ordas, L. Guillaume-Sage, and P. Maurine. 2015. EM injection: Fault model and locality. *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)* 00 (2015), 3–13. DOI: http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/FDTC.2015.9
[41] S. Ordas, L. Guillaume-Sage, K. Tobich, J.-M. Dutertre, and P. Maurine. 2015. *Evidence of a Larger EM-Induced Fault Model*. Springer, 245–259. DOI: http://dx.doi.org/10.1007/978-3-319-16763-3_15
[42] D. Page and F. Vercauteren. 2006. A fault attack on pairing-based cryptography. *IEEE Trans. Comp.* 55, 9 (Sept. 2006), 1075–1080. DOI: http://dx.doi.org/10.1109/TC.2006.134
[43] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont. 2016. Lightweight fault attack resistance in software using intra-instruction redundancy. In *Selected Areas in Cryptography (SAC)*.
[44] G. Ramalingam. 1999. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.* 21, 2 (March 1999), 175–188. DOI: http://dx.doi.org/10.1145/316686.316687
[45] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. 2005. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*. 243–254.
[46] Jörn-Marc Schmidt and Michael Hutter. 2013. The temperature side channel and heating fault attacks. In *CARDIS*.
[47] N. Timmers, A. Spruyt, and M. Witteman. 2016. Controlling PC on ARM using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 25–35. DOI: http://dx.doi.org/10.1109/FDTC.2016.18
[48] M. Werner, E. Wenger, and S. Mangard. 2016. Protecting the control flow of embedded processors against fault attacks. In *Smart Card Research and Advanced Applications: 14th International Conference, CARDIS*. 161–176. DOI: http://dx.doi.org/10.1007/978-3-319-31271-2_10
[49] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont. 2016. Software fault resistance is futile: Effective single-glitch attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 47–58. DOI: http://dx.doi.org/10.1109/FDTC.2016.21

Received May 2017; revised August 2017; accepted September 2017