Minimizing the Impact of Loop Hardening on Runtime Performance

Introduction

(a) The Problem

Fault attacks are a type of software attack in which an attacker with physical access to a device disrupts the execution of that device, leading to an alteration of behavior and, with the right targeting, potentially leaking sensitive information into the outside world. With the continuing proliferation of Internet-of-Things devices, attack surfaces for fault attacks have never been so prevalent. Unfortunately, some devices in the Internet-of-Things must operate in public spaces where the ability of device operators to provide physical security is necessarily limited. This means it is necessary to build fault-protection mechanisms into these devices, ideally at both the software and hardware levels. At the software level, one critical property to secure is to ensure that loops iterate the correct number of times. One example which illustrates the importance of this property is the Advanced Encryption Standard (AES) protocol: prior work has shown that disrupting iterations by corrupting the loop counter enables an attacker to retrieve the key, thus giving the attacker access to potentially sensitive data [1].

(b) Our Approach

To address this issue, we implement the work of Proy et al [2]. They implement an algorithm which performs minimal loop hardening. In loop hardening, instructions in the loop are duplicated so that disruptions to execution can be detected by comparing the results of the duplicated instructions. Discrepancies can be dealt with by directing control flow to an error handler. If done naively, that is, by duplicating every instruction in the loop, this process can be very expensive. The work of Proy et al. performs minimal loop hardening: that is, it performs

loop hardening that duplicates only the instructions absolutely necessary to ensure the correct number of loop iterations.

Instruction duplication causes instructions that are largely independent of the original instruction to be inserted into the basic blocks of the loop. In fact, there are no true (dataflow) dependencies between the duplicated instructions and the original instructions. We recognize this as an opportunity for parallelism through superscalar processing on multiple functional units. To take advantage of this opportunity, we implement a list scheduler to order instructions so that the hardware can execute them in parallel, thus reducing the runtime performance impact of loop hardening.

(c) Related work

In practice, according to Proy et al. [2], the most common strategy for performing software hardening is to have engineers insert protections manually, which is a time-consuming and error prone process. However, there has been other work in hardening software against fault attacks. Barenghi et al. [3] provide a specific hardening scheme for protecting AES in particular. However, their technique does not protect against register corruption. By duplicating instructions at the IR level, Proy et al.'s technique ensures that duplicated instructions each have their own register, meaning that it does not have this vulnerability. De Keulenaer et al. [4] perform loop hardening using executable binaries as input. Their technique is limited in the loops it can harden because it only works on loops whose counters are updated once per iteration. Oh et al. [5] , targeting hardware faults rather than fault attacks, proposes a redundancy approach that duplicates all instructions rather than just the minimal set of instructions necessary to ensure correct loop iteration count as done here.

(d) Contributions

Our contributions include

- An implementation of Proy et al.'s [2] minimal loop hardening set of algorithms
- A list scheduler which takes advantage of the superscalar processing opportunities that occur with loop hardening
- An evaluation of the performance impact of both the loop hardening and the loop hardening with list scheduling on a variety of real-world loops pulled from the coreutils set of utility programs.

Loop Hardening - Details

Loop hardening involves duplicating instructions inside the loop. The duplicated instructions perform some loop computations in parallel. In order to be useful, the results of these duplicated instructions must be compared with those of the original instructions; if there are discrepancies, this means that there is possibly an ongoing fault attack and that execution must be terminated and the program must exit or at minimum stop processing sensitive data that could be leaked.

Proy et al. focus on ensuring that the number of loop iterations is correct. This is important because for a number of cryptographic applications, an incorrect number of iterations can result in information being leaked and security guarantees being compromised. The algorithm is split into two main stages. The first identifies which instructions to duplicate, while the second performs the duplication.

(a) Identifying Instructions to Secure.

The first stage identifies which instructions to duplicate (secure). To do this, their algorithms identify each variable that is used to control a condition that leads out of the loop. For each of these variables, their algorithms perform a backwards slice through the loop's control-flow graph, recording the instructions that may be used to compute a given value. Allowing for arbitrary control flow in the loop introduces additional complexity: values used in loop conditions

may be updated conditionally. Therefore, it is also necessary to secure the control-flow instructions that determine whether or not a loop-controlling value is updated.

The algorithm which identifies instructions to secure is a worklist algorithm. Starting with a loop exit condition, each instruction's arguments are added to the worklist. This in turn allows all instructions that compute a value affecting whether the loop's exit to be processed. These values are recorded, and later duplicated. Branch instructions to duplicate, which may be necessary in case a loop-controlling value is dependent on control flow, are tracked separately because they require different handling to be duplicated correctly (see subsection (b)).

(b) Duplicating Instructions

In general, instruction duplication is relatively straightforward. A key caveat is that instructions whose arguments are the values produced by other duplicated instructions must use the values produced by the duplicates rather than the originals. However, the instruction can be simply duplicated and added to the same basic block as the original

Secured branch instructions, however, require special consideration. Duplicating a branch instruction and placing it directly after the original violates the abstraction rules of a basic block, and, in a related way, wouldn't serve to confirm that the branch is taken correctly. Instead, for each ith successor bb_s a basic block bb with a duplicated branch instruction, a new basic block bb_{new} with the duplicated branch instruction is inserted in place of that successor. All successors of bb_{new} are an error handler block except for the ith successor of bb_{new}. The ith successor of bb_{new} is bb_s, the original i^{th} successor of the old basic block. In this way, both a branch instruction and its duplicate must execute in the same way for control to flow to bb_s as normal. Otherwise, control will flow to the error handler. This strategy allows for the early detection and handling of inconsistent results between the original and duplicate dataflow subgraphs, which could potentially be the result of a fault attack.

The error-handler block is generated and inserted by the pass. It can hold arbitrary error-handling code; in our implementation, we simply call the “exit” function with a non-zero error code.

List Scheduling

The instructions duplicated in the previous step are, by design, entirely independent of the originals. This means that the duplicated instructions can be run in parallel with the originals. This is possible using the superscalar capabilities of modern processors—at least if there are enough functional units available for the types of instructions duplicated. In order to take advantage of this opportunity, we wrote a list scheduler that reorders instructions such that instructions near each-other in order are more likely to be independent. This allows the hardware’s out-of-order execution capabilities to be leveraged, since independent instructions will be fetched together more often, more closely in time, and thus hardware may run them simultaneously more frequently, therefore decreasing execution time. This list scheduler runs on LLVM IR; its purpose is to reorder instructions so that the hardware’s look-ahead naturally encounters independent instructions more often.

The list scheduler is based off of the list scheduling algorithm presented in class. First, we implemented a dependency DAG that is built for each basic block, with nodes representing instructions and edges representing dependencies. We add two types of dependency edges to the DAG. One type of edge is (true) dataflow dependency edges. We walk the use-def chains in the instructions, adding dependencies between the definitions and uses. The other type of edge we add are side-effect edges. The order of instructions with side effects must be preserved relative to other instructions with side effects in case the order of those side effects is important for the intended behavior of the program (e.g. two print calls occuring in the correct order). Thus, we add edges between subsequent occurrences of instructions with side effects. We need not add edges between an instruction with side effects and all subsequent instructions with

side effects because the dependencies are captured transitively in the DAG. We also ensure that memory reads (which do not, strictly speaking, have side effects) are included in this category; if a read instruction at a memory cell is reordered before a store instruction that also stores to that memory cell, incorrect behavior could result.

With the dependency graph built, we compute the priorities of each instruction. Starting from instructions that have no instructions dependent on them, we traverse the DAG, recursively computing the priority values of each instruction based on the cumulative maximum priority of the instructions that that instruction is dependent on.

Scheduling occurs in the typical way, with a ready list containing all instructions that have all of the dependencies satisfied, and an in-flight list that contains all instructions that have yet to finish executing. When an instruction finishes, we check each instruction dependent on it to see if those instructions have no other outstanding dependencies; if so, we add them to the ready list as well, ensuring the ready list is kept in decreasing priority order. There is no need to handle anti-edges in this formulation because each definition has a unique value, as enforced by SSA.

The basic block’s terminating instruction is dealt with separately. It is not added to the DAG, and is necessarily kept at the end of the block. Similarly, Phi nodes, for simplicity, are not added to the DAG either; rather, they are prepended to the start of the basic block.

Experimental Setup

In order to evaluate the performance of our replicated transformation, we focus on the GNU implementation of coreutils, since the benchmark suite of Proy et al. [2] comprises this. Moreover, since the problem we seek to address is largely concerned with edge computing devices an attacker may have physical access to, the expectations around hardware architecture are slightly different than for many consumer products specifically due to the commonality of relatively low-cost ARM microcontrollers in these devices. For this reason, we

performed the benchmarks on an emulated Cortex A53, using QEmu. The Cortex A53 is a 2-way superscalar processor with an 8-stage pipeline, implementing the Armv8-A ISA [6].

To accomplish this, we compile several different versions of the GNU coreutils, all of which are produced by using the coreutils' build system to produce LLVM bitcode which we then run several passes on, re-inserting the modified bitcode back into the coreutils build before completion. Specifically, we evaluate three different versions of the final binaries: (1) a "baseline runtime" version, produced by taking the bitcode, registering data dependencies between sequential instructions (to provide a comparison for later scheduling optimisations), and compiling a final binary; (2) a "hardened" version, produced by performing the same followed by the various loop hardening transformations discussed above; and (3) a "hardened with list scheduling" version, which comprises the same but with a final list scheduling pass.

We then selected a subset of coreutils, excluding some which are not possible to reasonably benchmark (e.g. because they inherently rely on timing, like `timeout`). In all, we tested 43 different coreutils programs, compiling them for Armv8-A, running them under QEmu simulating a Cortex A53 processor.

Experimental Evaluation

As we might expect, based on the fact that our programs are simply doing more work, by and large, we generally observe a slowdown relative to baseline performance. While we see some exceptions to this (e.g. `chown`), a major reason for these discrepancies is likely tied to the overhead a utility may experience on start-up: these are all short-running utilities which mostly comprise printing out minor information to the user. Major differences here are likely due to loop transformations required as a corollary of the loop hardening.

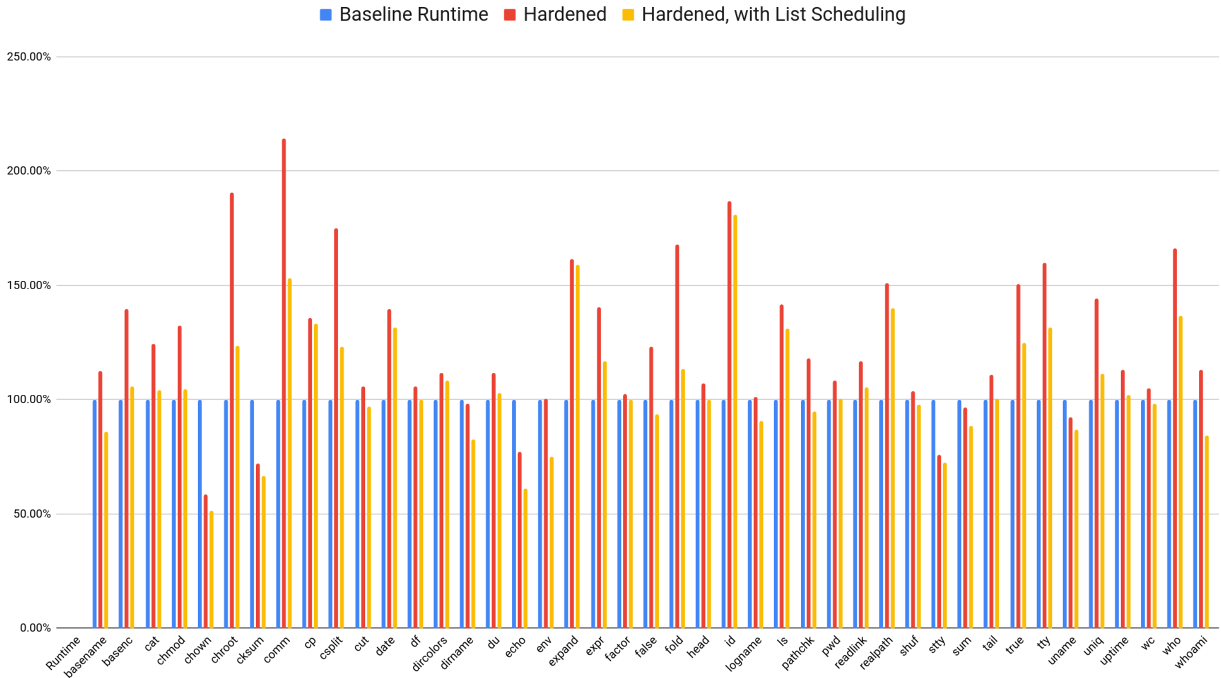

Figure 1: Comparison from baseline with all coreutils programs

A longer running example, which is more representative of the performance impact of our overall transformations is du. The benchmark for du was on the longer running side, approximately 30 seconds per-run of the utility, so the results are less subject to the same noise as short-running programs.

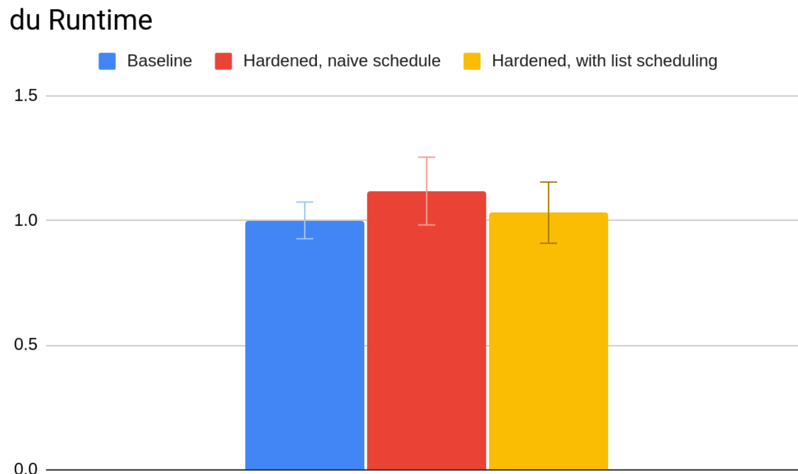

Figure 2: Comparison for du, with error

Here, we see that while both hardened versions of the program are slower running than the baseline version, we have a significant difference between the naive placement of instructions compared to the list scheduled version. Moreover, despite the redundancy in instructions, performance is largely comparable. A major contributor to this result is likely the fact that our instruction duplication results in inherently non-data dependent instructions which have the same control dependencies; this leads to more exploitable parallelism for the processor to take advantage of when performing out of order execution.

Surprises and Lessons Learned

Originally, we anticipated that we may see a more significant performance penalty associated with the duplicated instructions, and perhaps more significantly, the duplicated loops. However, consistent with Proy et al. we see, in general, a modest performance penalty, rather than a significant one.

The largest lesson which can be learned from this is the true capability of modern processors, even relatively modest ones, to exploit instruction-level parallelism. Since the generated programs are redundant, but largely consist of independent copies of original

instructions (to the extent that new instructions are inserted), there exist myriad opportunities to exploit data-parallelism in the final code. Likewise, modern branch prediction likely helps us significantly in reducing the runtime cost of duplicated branch instructions, since simply predicting the same branch seen is essentially always correct, barring modifications to the instructions or data at runtime by external causes.

Conclusions and Future Work

In this paper, we addressed the problem of fault attacks on Internet-of-Things (IoT) devices by implementing Proy et al.'s algorithm for minimal loop hardening. Our approach involves duplicating only the necessary instructions to ensure the correct number of loop iterations and taking advantage of the opportunity for parallelism through superscalar processing on multiple functional units. We implemented a list scheduler to order instructions for parallel execution, reducing the runtime performance impact of loop hardening. Our contributions include an implementation of Proy et al.'s algorithm, a list scheduler, and an evaluation of the performance impact on GNU coreutils. Largely, performance is minimally impacted, at least when dealing with microcontrollers with fair pipelines and multiple functional units which can be concurrently utilized.

Possible future work at this stage consists largely of more realistic evaluation. More extensive testing for the runtime of transformed programs, namely security sensitive programs reflecting the possible concerns associated with the threat model discussed in the introduction would give a better representation of the real-world cost associated with the hardening techniques discussed. Likewise, expanded coverage of processors evaluated would give a more full-featured picture of the impact of these transformations on lower-end devices, which may only have 3-stage pipelines and lack the capability to meaningfully exploit instruction-level parallelism.

Along a different axis of evaluation, a real-world assessment of the robustness of these transformations against irradiation-based attacks would provide the ability for a better analysis of the security-runtime trade-off possible. While Proy et al. evaluate this somewhat, in the context of bit flips, this was done using a custom modification to gdb, rather than physical testing.

Distribution of Total Credit

We believe a 50%/50% distribution of credit is fair.

References

[1] Dehbaoui, A., Mirbaha, A. P., Moro, N., Dutertre, J. M., & Tria, A. (2013). Electromagnetic glitch on the AES round counter. In *Constructive Side-Channel Analysis and Secure Design: 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers 4* (pp. 17-31). Springer Berlin Heidelberg.

[2] Proy, J., Heydemann, K., Berzati, A., & Cohen, A. (2017). Compiler-assisted loop hardening against fault attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, *14*(4), 1-25.

[3] Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., & Regazzoni, F. (2010, October). Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security* (pp. 1-10).

[4] De Keulenaer, R., Maebe, J., De Bosschere, K., & De Sutter, B. (2016). Link-time smart card code hardening. *International Journal of Information Security*, *15*, 111-130.

[5] Oh, N., Shirvani, P. P., & McCluskey, E. J. (2002). Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, *51*(1), 63-75.

[6] ARM. (2014). *ARM Cortex-A53 MPCore Processor Technical Reference Manual.*, from https://developer.arm.com/documentation/ddi0500/d