UNIVERSITY OF CALIFORNIA,
IRVINE

SIMD Assisted Fault Detection and Fault Attack Mitigation

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Zhi Chen

Dissertation Committee:
Professor Alex Nicolau, Chair
Professor Alex Veidenbaum
Professor Nikil Dutt

2018
DEDICATION

To my parents, wife, daughter, and son for their unconditional love and support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>viii</td>
</tr>
<tr>
<td>CURRICULUM VITAE</td>
<td>ix</td>
</tr>
<tr>
<td>ABSTRACT OF THE DISSERTATION</td>
<td>xi</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Feasibility Study</td>
<td>10</td>
</tr>
<tr>
<td>2.1 The approach</td>
<td>12</td>
</tr>
<tr>
<td>2.1.1 Vectorization with intrinsics</td>
<td>12</td>
</tr>
<tr>
<td>2.1.2 Error detection</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Experimental Evaluation</td>
<td>19</td>
</tr>
<tr>
<td>2.2.1 Evaluation</td>
<td>21</td>
</tr>
<tr>
<td>2.2.2 Energy consumption</td>
<td>25</td>
</tr>
<tr>
<td>2.3 Summary</td>
<td>27</td>
</tr>
<tr>
<td>3 Compilation Flow</td>
<td>29</td>
</tr>
<tr>
<td>3.1 Compiler Implementation</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1 Arithmetic Instructions, Logical Instructions, and Bit-wise Instructions</td>
<td>32</td>
</tr>
<tr>
<td>3.1.2 Memory Instructions</td>
<td>33</td>
</tr>
<tr>
<td>3.1.3 Branch Instructions</td>
<td>35</td>
</tr>
<tr>
<td>3.1.4 Call Instructions</td>
<td>36</td>
</tr>
<tr>
<td>3.1.5 Other Instructions</td>
<td>37</td>
</tr>
<tr>
<td>3.1.6 Multibit Errors</td>
<td>38</td>
</tr>
<tr>
<td>3.2 Overall Compilation Process</td>
<td>39</td>
</tr>
<tr>
<td>3.2.1 Instruction Duplication</td>
<td>39</td>
</tr>
<tr>
<td>3.2.2 Scalar Instruction Duplication</td>
<td>43</td>
</tr>
<tr>
<td>3.2.3 SIMD-based Duplication</td>
<td>45</td>
</tr>
<tr>
<td>3.3 Error Checking</td>
<td>48</td>
</tr>
<tr>
<td>3.4 Error Correction</td>
<td>50</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
</tr>
<tr>
<td>4</td>
<td>Experimental Evaluation</td>
</tr>
<tr>
<td>4.1</td>
<td>Experimental Setup</td>
</tr>
<tr>
<td>4.2</td>
<td>Instruction Duplication Only</td>
</tr>
<tr>
<td>4.3</td>
<td>Instruction Duplication with Error Checking</td>
</tr>
<tr>
<td>4.4</td>
<td>Code Size Overhead</td>
</tr>
<tr>
<td>4.5</td>
<td>Overhead with Memory Protection</td>
</tr>
<tr>
<td>5</td>
<td>Mitigation of Crypto Fault Attacks</td>
</tr>
<tr>
<td>5.1</td>
<td>Fault Model</td>
</tr>
<tr>
<td>5.2</td>
<td>Evaluation</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Experimental Setup</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Fault Coverage</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Fault Attack Prevention Discussion</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Fault Detection Overhead</td>
</tr>
<tr>
<td>6</td>
<td>Related Work</td>
</tr>
<tr>
<td>6.1</td>
<td>Hardware level</td>
</tr>
<tr>
<td>6.2</td>
<td>Software level</td>
</tr>
<tr>
<td>6.3</td>
<td>Hybrid approaches</td>
</tr>
<tr>
<td>6.4</td>
<td>Fault Attacks and Mitigation</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions and Future Work</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusions</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
</tr>
</tbody>
</table>

Bibliography | 96
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Performance speedup with vectorization on difference benchmarks using ICC and LLVM, respectively.</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>A code snippet from Aobench.</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Vectorization using AVX intrinsics with loop splitting for ambient Occlusion function in Aobench.</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Scalar mode error detection for the loop in Aobench. For the sake of brevity, the duplication of other variables is not shown in this figure.</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Error detection and correction for the loop in the Aobench.</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Vector mode duplication for error detection. YMM registers are used by AVX instructions. All YMM registers in this figure are 256 bits, e.g. ymm3 and ymm4 contain 4 double f.p. values.</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Part of AVX assembly code from Aobench.</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>Slowdown of instruction duplication for: (a) SSE intrinsics over the baseline, (b) scalar and vector instruction duplication.</td>
<td>22</td>
</tr>
<tr>
<td>2.8</td>
<td>Speedup of vector mode instruction duplication over the baseline with instruction duplication.</td>
<td>24</td>
</tr>
<tr>
<td>2.9</td>
<td>Energy reduction of the SSE and AVX versions over the baseline.</td>
<td>25</td>
</tr>
<tr>
<td>2.10</td>
<td>Energy increase of instruction duplication.</td>
<td>26</td>
</tr>
<tr>
<td>2.11</td>
<td>Energy reduction of error detection using vector mode instruction duplication.</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Scalar (a) and SIMD-based (b,c) source code duplication and error checking. (b): codes without SIMD instructions and (c): codes with SSE instructions.</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Memory instruction duplication using (a) gather, (b) scatter.</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Compilation flow for our fault tolerant code generation approach.</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>Instruction duplication example. (a) Source code, (b) Partial LLVM bitcode.</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>Transformed LLVM bitcode for the LLVM bitcode in Figure 3.4(b). A highlighted IR instruction is a duplicate, followed by the original instruction.</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>LLVM IR after SIMD-based instruction duplication.</td>
<td>44</td>
</tr>
<tr>
<td>3.7</td>
<td>(a) A basic block with 4 instructions. (b) The basic block after SIMD-based instruction duplication. (c) Error checking after each duplicated instruction, (d) Error checking at only before a store.</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Performance slowdown of scalar and vector instruction duplication (no error checking added).</td>
<td>53</td>
</tr>
</tbody>
</table>
4.2 Energy overhead of scalar and vector instruction duplication (no error checking). 53
4.3 Performance slowdown for instruction duplication with selective and full error checking. 57
4.4 Energy overhead of instruction duplication with selective and full error checking. 58
4.5 Binary size of scalar and vector instruction duplication with selective and full error checking. 60
4.6 Performance slowdown of scalar and vector instruction duplication with load and store protection. 62
4.7 Energy overhead of scalar and vector instruction duplication with load and store protection. 63
4.8 Instruction count overhead of scalar and vector instruction duplication with load and store protection. 63

5.1 The breakdown of faults injected on each type of instruction for different mechanisms: no fault detection (N), duplication without gather and scatter instructions (O), duplication with gather and scatter instructions (W). 71
5.2 Fault coverage by each cryptographic algorithm among different mechanisms: no fault detection (N), duplication without gather and scatter instructions (O), duplication with gather and scatter instructions (W). 74
5.3 The slowdowns of error detection enabled cryptographic algorithms to compute private and public keys compared to the original ones without error detection for different input sizes. 77
5.4 The dynamic instruction count overhead for duplication with and without gather/scatter instructions. 79

6.1 Signature based control flow protection. $G = s_2$ at the dest would hold if the control flow transfers correctly. 89
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>System configurations</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Profiling information of each benchmark</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>The execution time (s) and energy consumption (J) of baseline and baseline</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>with duplication</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Standard deviation of performance and energy overheads</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of fault coverage rates of each category among no fault detection</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>(N), duplication without gather and scatter (O), and duplication with gather</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and scatter (W)</td>
<td></td>
</tr>
</tbody>
</table>
I would like to thank my advisors, Prof. Alex Nicolau and Alex Veidenbaum. Without their continuous enlightenment and tremendous help which includes giving me research advice and financial support, this dissertation wouldn’t even have been possible. I would like to express my sincerest and deepest thanks to them for keeping my spirits up no matter what the circumstances. It has been my great honor to join Prof. Nicolau and Prof. Veidenbaum’s research group as PhD student where I’ve learned how to find problems and solve them efficiently. In addition, during the past five years, they have instilled the appropriate way to write papers and collaborate with other people in my mind. I was also encouraged to participate conference and events and talk to various researchers to present our work, which improved my presentation and communication skills.

I would like to thank Prof. David Padua at the University of Illinois Urbana-Champaign for providing me immerse help and guidance on the vectorization project. Without his constructive comments, we would not be able to build the LORE repository. A very special gratitude also goes out to Dr. David Wong at Intel Corp., Dr. Zehra Sura at IBM Research, and Prof. Josep Torrellas who gave me valuable comments during each meeting and polished my paper.

I would like to thank my thesis and candidacy committee: Prof. Nikil Dutt, Prof. Guoqing Harry Xu, Prof. Mohammad Al Faruque for their insightful comments and encouragement which helped me improve my work.

I would like to thank Dr. Rosario Cammarota and Dr. Nahid Farhady Ghalaty for their insightful discussion on the CAMFAS project. They have been regular organizing meetings to discussion possible directions to apply the SIMD-based error detection and fault attacks approach.

I would like to thank my colleagues and project partners, Sajjad Taheri, Dr. Fausto Artico, Neftali Watkinson, Aniket Shivam, Gongjin Sun, Junjie Shen, Zhangxiaowen Gong, Justin Szaday for the stimulating discussions and for the nights and days we were working together before project and/or paper deadlines. I was great sharing the lab and working on the projects with you during the past five years.

Last but by no means the least, this list cannot be complete without thanking my wife Jing Li, my children, my parents, and my siblings for their unfailing support.

The work presented in this dissertation has been supported by NSF grants
CURRICULUM VITAE

Zhi Chen

EDUCATION

Doctor of Philosophy in Computer Science 2018
University of California, Irvine

Master of Science in Electrical Engineering 2013
University of Kentucky

Master of Science in Computer Science 2011
Hunan University

Bachelor of Science in Computer Science 2008
Huaihua University

RESEARCH EXPERIENCE

Graduate Research Assistant 2013–2018
University of California, Irvine

TEACHING EXPERIENCE

Teaching Assistant 2014–2016
University of California, Irvine

PROFESSIONAL EXPERIENCE

Research Intern Summer 2012
Oak Ridge National Laboratory

Research Intern Summer 2014
Broadcom Corp.

Software Engineer Intern Summer 2016
Facebook Inc.

Software Engineer Intern Summer 2017
Google Inc.
REFEREED PUBLICATIONS

LORE: A Loop Repository for the Evaluation of Compilers
October 2017
2017 IEEE International Symposium on Workload Characterization (IISWC)

CAMFAS: A Compiler Approach to Mitigate Fault Attacks via Enhanced SIMDization
September 2017
2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)

SIMD-based soft error detection
May 2016
Proceedings of the ACM International Conference on Computing Frontiers (CF)

Software fault tolerance for FPUs via vectorization
July 2015

SOFTWARE

LORE repository
http://www.vectorization.computer/

A loop repository to evaluate compilers.
Modern processors continue to aggressively scale down the feature size and reduce voltage levels to run faster and be more energy efficient. However, this trend also poses significant reliability concern as it makes transistors more susceptible to soft errors. Soft errors are transient. Although they don’t impair the computing systems permanently, these errors can corrupt the output of a program or even crash the entire system. Hardware or software redundant techniques could be used to detect errors during the execution of a program. However, hardware redundancy, e.g. DMR (dual-modular redundancy) and TMR (triple-modular redundancy), leads to significant area overhead and very high energy cost. Software redundancy, e.g. instruction duplication, has lower performance and energy penalty and virtually no hardware cost by sacrificing a small degree of error coverage. Yet commodity processors generally don’t require “five-nines” reliability as they are not mission-critical. Instead, performance and energy consumption have more priority. This dissertation proposes a novel approach to instruction duplication, which exploits the redundancy within SIMD instructions. The key idea is to pack the original data and its duplicate in the different lanes of the same vector register instead of executing two scalar instructions separately as these registers are underutilized on most applications. The proposed solution is implemented in the LLVM compiler as a stand-alone pass. Evaluation on a host of benchmarks reveal that proposed SIMD-based error detection technique causes much less performance, code size,
and energy overheads.

This dissertation further extends the proposed approach as a countermeasure to protect cryptographic algorithms. These algorithms are widely adopted in modern processors and embedded systems to protect information. A number of popular cryptographic algorithms in the Libgcrypt library are protected using the SIMD-based instruction duplication technique. A large amount of errors are injected to these algorithms. The results show that almost all injected faults can be detected with reasonable performance and code size cost.
Chapter 1

Introduction

Error/fault detection is fundamental for a fault tolerant system since any fault has to be detected and handled before the system can continue. It has been extremely researched. Numerous error detection and correction mechanisms have been proposed to cope with different types of errors and faults during operation. These techniques are usually applied to various applications or scenarios by trading off performance, fault coverage, chip area, and energy, etc [49].

Two types of errors are well studied and understood in the literature, namely hard errors (HE) and soft errors (SE). Hard errors are generally caused by permanent errors in a circuit, e.g. design bugs or process defects. This type of errors can be expected to occur repeatedly in the same way after restarting the machine because they are permanent stuck-at faults that occur in the circuit [46]. Therefore, the most straightforward way to handle them is hardware replacement.

On the other hand, soft errors are transient errors in computer hardware caused by electrical noise and high-energy particle strikes caused by chip packaging impurities and cosmic radiation, e.g. alpha particles and high-energy neutrinos. The impact of such a particle is of
a very short duration, is very localized and happens in isolation, i.e. multiple simultaneous soft errors are very unlikely. Thus another term for such errors is Single Event Upsets. These errors can cause bits of storage elements to flip or disrupt the operation of a combinational logic circuit [99]. This may corrupt the output of an application or even crash a computer system. SRAM cells are very prone to such errors and therefore have been hardened via circuit design and error checking codes (ECC) to improve resilience. It was estimated in prior research that SERs for SRAM will remain roughly constant over several technology generations, at $10^{-4}$ FIT/bit [111, 99]. Latches and combinational logic have so far not been a problem, but their transient errors are also increasing [95, 101, 109]. It is expected that faults in combinational logic will be roughly on par with errors in unprotected SRAM arrays [95, 35]. Therefore, many-core chips and large systems built from them will need to be protected from such errors. Technology scaling with its reduced feature size and lower supply voltage achieves better processor performance and energy efficiency. It also leads to variation in process, voltage, and temperature. The variation in turn makes computer systems more susceptible to soft errors, posing significant reliability challenges [66, 115, 17, 10].

Previous work has proposed both hardware and software solutions to soft errors via periodically checking processor and memory states. Upon the detection of a soft error, the system could initiate recovery (i.e. rollback to an recorded fault-free state) or exit the execution directly. The hardware techniques that can detect and correct soft errors are based on hardware redundancy, e.g. dual modular redundancy (DMR) and triple modular redundancy (TMR). DMR and TMR offer extremely high availability. However, these forms of modular redundancy require significant chip area, have performance and energy overheads. These costly approaches, hence, are only used for mission-critical applications, they are too expensive for commodity processors where occasional errors are not a concern [45, 52]. Thus software implemented fault tolerance is the only alternative for such processors where the overhead of the full hardware duplication is prohibitively expensive. Software solutions often attempt to maintain a relatively low overhead in performance by sacrificing some degree of
reliability while not entirely ignoring it. One such software solution is instruction duplication. Each instruction in a program is executed twice and the two results are compared. The probability that both have been affected by the same soft error is negligible and thus the comparison will detect a soft error.

Instruction duplication and the addition of comparisons to check the outputs of the two instructions can be implemented in a compiler and thus fully automated. Techniques such as EDDI [73] and SWIFT [87] utilize instruction level parallelism (ILP) to overlap the execution of an original instruction and its duplicate. However, instruction duplication has a significant performance overhead. EDDI provided high coverage but required the program memory to be duplicated. It loaded input operands twice, one from each copy of the program memory. Two stores were performed, one to each copy. SWIFT reduced the overhead by a) not duplicating the memory and b) by performing checks less frequently (i.e. not for every duplicated instruction). The former reduced coverage compared to EDDI as stores cannot be duplicated. Still, the overhead remained significant.

We performed a feasibility study of a different software solution [20] to instruction duplication using vectorization. Instead of replicating an original instruction we replicated its operands using SIMD registers and performed an SIMD operation for duplicated execution. Low performance overheads, e.g., only a 1.2x slowdown, was achieved.

The feasibility study focused on floating point computations in application kernels and manually inserted SSE/AVX intrinsics at the source level for duplication. Similar to SWIFT, we did not duplicate memory. Integer instructions were not considered. The results of the study indicated that the overhead of full duplication using SIMD instructions was quite low but checking on every instruction was still expensive.

Overall, it is more appropriate to think of the SIMD-based instruction duplication as data duplication since only one SIMD instruction is sufficient for duplicated execution. This pro-
Figure 1.1: Performance speedup with vectorization on difference benchmarks using ICC and LLVM, respectively.

Figure 1.1: Performance speedup with vectorization on difference benchmarks using ICC and LLVM, respectively.

provides several advantages compared to the scalar version of instruction duplication: reduced code size and register pressure, better performance, and easier compilation. For example, the high register pressure of the scalar duplication used in the traditional software techniques is very likely to increase the chance that a register will be spilled on memory. As a result, the compiler may select complex multi-memory access operations instead of the simple loads and stores [27]. These complex operations (e.g. \textit{ldm} and \textit{stm}) are more susceptible to soft errors than the simple ones or register-register operations [102]. However, vector instruction duplication would lead to a much lower register pressure as the original instruction and its duplicate essentially share the same register by using spatial redundancy rather than temporal redundancy. Vector instructions can also be much more energy efficient because only one instruction is required to be fetched per data operation[92]. However, additional instructions may be needed, for instance to replicate the data in a vector register and to deal with other idiosyncrasies of SIMD instruction sets.

The goal of this dissertation is to automatically generate and optimize SIMD-based instruction duplication in a compiler. It focuses on soft error detection. Soft errors are infrequent
enough to allow correction to be relatively slow. Detection, on the other hand, has to be
continuous and thus fast. The duplication is accomplished by using SIMD vector instruc-
tions. SIMD instructions are available today in most commercial processors for both integer
and floating point data types. They operate on separate, wider registers that fit multiple
operands. This allows operands of the original instruction and its duplicate to be packed in
the same SIMD register and instruction “duplication” is replaced by execution of a single
SIMD instruction. Error checking still includes scalar instructions, in particular a branch in
the case of error detection.

A soft error can affect not only computational instructions, comparisons, and memory in-
structions, which our approach checks, but also many other parts of a processor that instruc-
tion duplication cannot check. These include instruction decoding, branch target address
computation (without using signatures), memory address computation (without duplicating
loads), store errors (without duplicating the memory content), etc. Prior work on instruc-
tion duplication thus also could not check decoder and other hardware errors “invisible” to
instructions.

Compiler instruction duplication has to address several issues. First, one needs to decide
at what point in the compilation process to perform the duplication. If performed before
optimizations are applied, the duplicated instructions will probably be dead code eliminated.
Therefore, the dead code elimination pass may have to be modified to fulfill this task. It also
cannot be performed after register allocation. Otherwise, one has to handle the allocation
of duplicated registers which requires a large amount of work and is quite error prone. The
right place depends on a compiler infrastructure used. In our framework, the duplication is
performed at the intermediate representation level where all the operations are visible for
duplication and register allocation and other optimization are left to the compiler back-end.
Additional registers needed for duplicated instructions may increase register pressure which
is another issue to be solved.
Another issue is errors in the instructions added to detect errors. Similar to prior work on software error detection through duplication, e.g. [44, 73, 87] etc, we will not duplicate and check the added error checking code. This dissertation assumes a single error at a time only (SEU), so an operation error and an error in the checking code error cannot both happen at the same time. An error in the error checking code, e.g. comparison or vector sign extension, is not fatal by itself. All that will happen is that a fault will be “detected,” but it can be corrected by the error correction mechanism in place. This will also be a very infrequent event. Even in the case of multiple errors, the probability that both an instruction and its error checking code have errors at the same time is negligible.

We designed a compilation approach for SIMD-based instruction duplication using the LLVM compiler infrastructure [48]. Duplication is performed at the intermediate representation (IR) level, after standard transformations and optimizations have been performed but before register allocation. This allows the register allocator to deal with original and duplicated instructions simultaneously. In fact, using SIMD registers allows us to duplicate operands without allocating additional registers. One question that can be asked at this point is, what if the code already has SIMD instructions? Let us look at the performance of vectorized programs to answer this question.

Figure 1.1 shows the speedup achieved by auto-vectorization for ICC and LLVM for benchmarks in this study. The auto-vectorization improves the performance of these applications by an average of 12% and 1% on ICC and LLVM, respectively. This shows that the SIMD units are usually underutilized. Note that these benchmarks are f.p. benchmarks, which are expected to have better potential for vectorization than integer benchmarks. Also note that even benchmarks with high speedup from vectorization, e.g. Linpack, may still have underutilized vector resources as the compiler may decide to not generate code to use the widest register available on a processor. This will be seen in the low SIMD duplication overhead for the Linpack benchmark. Therefore, the vector units might be idle or only a few lanes of
them are occupied. It provides an evidence that a plenty of vector resources are available to achieve the redundancy purpose “for free”.

Vector devices are not only available on commodity processors, they are also widely deployed on embedded processors nowadays. Most embedded processors adopt public-key cryptography schemes to secure or authenticate critical data, such as online banking account. Typical examples of consumer devices equipped with such cryptographic algorithms are smart phones, smart watches, and Blu-ray players, etc. On the other hands, these devices are also common victims of fault attacks. For instance, an adversary may induce faults into such a device during its execution of a program. The adversary can observe the output of the program and correlate the faulty result of a computation to an algorithm. The comparison between the correct output and the faulty output may lead to the leakage of the secret key.

The severity of fault attacks on general purpose microprocessors directly correlates with the increasing importance or criticality of the contents being processed. For example, premium contents and payments are processed on mobile handsets and servers; metering information is processed on “relatively” resource constrained devices that use increasingly sophisticated System on Chips (SoCs). In both cases the “valuables” are protected by encryption and access control mechanisms which are prone to physical attacks, i.e., side-channel and fault attacks [60].

In a fault attack (FA), an attacker injects faults into the underlying processor hardware to temporarily alter the execution of instructions. For example, the attacker can use clock glitches to force the processor to run beyond its nominal operating conditions. Alternatively, an attacker can perform an electromagnetic injection to alter the current value of registers [7]. Applying perturbations to the microprocessor creates faulty data values upon which instructions operate, or it can change the instruction operands. Then, the attacker observes the fault effect in the output of the running software. Finally, the attacker can break the security of the system by performing a systematic fault analysis method, e.g.,
he/she can use Differential Fault Intensity Analysis on the observed output to perform key extraction attacks [30].

Instruction duplication countermeasures [9] are well established mitigations against fault attacks by assuming that either the original instruction or its duplicate will not be affected by the same faults. Based on the recent research proposed in [77], the higher the redundancy granularity, the more resistant it is against fault attacks. We extend the vector based instruction duplication technique to protect against fault attacks in cryptographic algorithms. Our vector assisted duplication effectively achieves spatial redundancy, which makes the pipeline based error injection indefeasible.

The contributions of this thesis are as follows:

- A new compilation approach using SIMD vector units for instruction duplication and error checking
- A compiler implementation of the proposed approach is presented and discussed.
- The duplication in the library function calls that is supported by LLVM with vector compatible prototypes
- An evaluation and comparison of the SIMD and other instruction duplication techniques on an Intel processor
- The application of the SIMD-based duplication to cryptographic algorithms to protect against fault attacks and mitigate fault attacks.
- The evaluation on Libgcrypt library shows the effectiveness of our mitigation technique through fault injection.

Overall, this work shows that the proposed approach has a relatively low overhead in performance, energy, and code size compared to scalar instruction duplication. It also offers a
relatively low-cost approach to protect cryptographic algorithms against fault attacks. This is the result of lower instruction counts and other efficiencies of SIMD execution.

The organization of the rest of this dissertation is as follows.

Chapter 2 describes the motivation of the SIMD-based error detection technique where a manual approach using intrinsics are employed to study the feasibility of SIMD-based duplication scheme.

Chapter 3 presents the compilation flow of the SIMD-based error detection approach. This chapter discusses how each instruction type will be handled and how error checking will be performed.

Chapter 4 evaluates the overhead of the proposed technique on modern processors.

Chapter 5 introduces the application of our SIMD-based approach on cryptographic algorithms to fight against fault attacks. Error injection will be also performed in this chapter to evaluate the efficiency of the proposed countermeasure.

Chapter 6 discusses the related work.

Finally, Chapter 7 concludes this dissertation and poses some future directions that can be looked into.
Chapter 2

Feasibility Study

This chapter presents a feasibility study of our SIMD based fault detection techniques. As we observed that the vector units are still underutilized in Chapter 1, the basic idea of our fault detection mechanism is to pack the original data and its duplicate in the same vector register which essentially transforms the traditional operation duplication into data duplication. The duplication doesn’t change the semantics of the original program, but it only moves the execution of the instructions from the scalar execution unit to vector units or from the narrower registers to wide ones.

There are generally three ways to achieve this type of duplication, I) programming in assembly, II) using SIMD intrinsics provided by most of platforms, and III) relying on the compiler to generate fault-tolerant code with vectorization. Coding in assembly is tedious and error prone because there are a great volume of SIMD instructions and they have more complicated sematics than the conventional scalar assembly code. Therefore, it requires a significant amount of programming efforts from experts. Generating vectorized code to detect errors is our ultimate goal. It needs to modify a compiler to generate the fault conscious code, but then the framework could apply to many other codes, instead of programming in
assembly for each of them. However, before modifying a compiler, we start to verify the idea using the SIMD intrinsics. Intrinsic functions are low level built-in functions that will be directly replaced with a corresponding assembly instruction by the compiler. Programmers have the flexibility to code them in a C/C++ way where they don’t need to specifically manipulate registers. The compiler will decide where to store an intrinsic variable and it preserves some room to optimize and/or transform the code written in intrinsics. Here, we only confine ourselves to arithmetic (particularly f.p. instructions) as f.p. execution can consume up to 75% of core power in compute-intensive applications [100]. We will present the full implementation in the compiler in the next chapter.

The rest of this chapter proposes to detect soft errors while reducing the time and energy overhead of executing redundant arithmetic instructions by using vector units. Most modern processors support SIMD instructions, such as Intel SSE and AVX. SSE/AVX can run 2/4 double precision or 4/8 single precision FP instructions simultaneously, using separate 128/256-bit register files. The vector unit thus provides the required redundant FPUs even if they are not available in the “scalar f.p. unit”. In fact, one can argue that since soft error detection should be always performed the “scalar f.p. unit” can be eliminated.

Therefore, one can detect soft errors by using duplicated data packed into vector registers and using vector instructions. Comparison of duplicated data can also be done in vector mode. For example, one can load two result values in a YMM register and broadcast them to the remaining two slots of the register. Soft error detection is then accomplished by comparing the results in the YMM register using vector comparison instructions.

Correction can be accomplished in a similar way by triplicating data in a vector register. However, it is slightly more expensive than detection due to an additional comparison required for a simple voting. This chapter advocates performing the third operation and comparison only if an error is detected. Soft errors are very short-lived, thus re-executing on the same f.p. unit is sufficient for getting the 3rd result. In addition triplicating all instructions would
utilize 3/4 of the vector bandwidth. Duplication uses only 1/2 and thus allows two instructions to be duplicated in one vector instruction. Thus the rest of this chapter will focus on detection only. The data from memory is assumed to be correct in this chapter.

Finally, the job of generating code for the vector unit does not need to be performed manually. This is a much simpler task than automatic vectorization and can be done in a compiler which will be the main subject of the next Chapter. For this chapter duplicated or triplicated vector code was manually added in critical f.p. loops of a set of benchmarks. This was done using SSE/AVX intrinsics in the hot loops of a benchmark. It can be argued that the execution time of compiled redundant code will be even lower as the compiler is able to optimize a loop in its entirety.

2.1 The approach

This section describes our vectorization and error detection techniques. The input is an FP application written in C/C++ and the final output is a vectorized program with instruction duplication.

2.1.1 Vectorization with intrinsics

The first step is to vectorize the selected applications. This was performed at the source level and for hot loops as determined by profiling. A hand compilation was performed at this point by replacing N iterations an original loop with a loop body comprised of vector intrinsics of length N. For example, Figure 2.1 shows a loop nest from function ambient_occlusion in Aobench [5], which accounts for 98% of the total execution time. It contains an unvectorizable function drand48. It was vectorized as shown in Figure 2.2 using loop splitting and SSE/AVX intrinsics.
Figure 2.1: A code snippet from Aobench.

The f.p. random number generator had to be split from the loop body because there is no intrinsic for the random function (Line 2-5). Vectorization cannot be performed when a library function is present. It is possible to use AVX intrinsic instructions to vectorize the `sqrt` and trigonometric functions. Unlike ICC, there are no direct SSE intrinsics instructions to vectorize trigonometric functions in the GCC library. One solution is to use ACML/MKL library from AMD/Intel. Similarly, we replace the rest of code with the corresponding AVX intrinsic instructions. Only vectorization through AVX intrinsics is presented here because SSE has corresponding instructions. The only difference is that SSE uses XMM registers (128 bits) but AVX uses YMM registers (256 bits). Finally, the inner loop is vectorized since we can fetch 4 elements from the arrays filled with random numbers and process them simultaneously. The advantage is that we only execute 1/4 inner loops of the original program therefore significantly reducing the execution time and energy consumption.
// loop splitting
for (j = 0; j < ntheta; j++) {
    for (i = 0; i < nphi; i++) {
        thRand[i] = drand48();
        phiRand[i] = drand48();
    }

    for (i = 0; i < nphi; i +=4) {
        theRnd = _mm256_load_pd(&thRand[i]);
        t = _mm256_sqrt_pd(theRnd);
        phiRnd = _mm256_load_pd(&phiRand[i]);
        twoPi = _mm256_set1_pd(2.0*PI);
        phi = _mm256_mul_pd(twoPi, phiRnd);
        sinPhi = _mm256_sin_pd(phi);
        cosPhi = _mm256_cos_pd(phi);

        x = _mm256_mul_pd(cosPhi, t);
        y = _mm256_mul_pd(sinPhi, t);
        ones = _mm256_set1_pd(1.0);
        z = _mm256_sqrt_pd(_mm256_sub_pd(ones,
                          _mm256_mul_pd(t, t)));

        xx0 = _mm256_mul_pd(x, 
                          _mm256_set1_pd(bas[0].x));
        yx1 = _mm256_mul_pd(y, 
                          _mm256_set1_pd(bas[1].x));
        zx2 = _mm256_mul_pd(z, 
                          _mm256_set1_pd(bas[2].x));
        rx = _mm256_add_pd(
                          _mm256_add_pd(xx0, yx1), zx2);
        // do the same to get ry and rz
    }
}
2.1.2 Error detection

The next step is to detect soft errors. We will demonstrate how scalar mode and vector mode instruction duplication work in the following two sections.

Scalar mode error detection

To protect the FPU from faults, we need to duplicate all f.p. operations. The effect of duplicating such operations are shown in Figure 2.3. Line 6, Line 20, and Line 27 are the duplicated instructions. For the purpose of demonstration, we only show a few duplicated instructions in this example. Then the resultant values are compared for equality with a checking instruction to detect the error. If an error is identified, an error correction process is initialized by executing the same instruction one more time for triplication and then performing voting to get the majority. The error is considered permanent if the counter reaches a predefined threshold and the FPU is disabled.

In addition simplicity, another advantage of this approach is that all soft errors could be detected since all f.p. instructions are protected. The disadvantage of this technique is the high cost. It may double the execution time and energy consumption because more than 2x instructions are required for duplication and comparison. The exact overhead depends on available processor ILP and instruction dependencies. To keep the advantage of scalar instruction duplication but reduce its high costs caused by redundant instructions, we provide vector mode error detection through parallelizing the execution of the original instruction and duplicated ones.
// error detection by inserting checkers
for (j = 0; j < ntheta; j++) {
    for (i = 0; i < nphi; i++) {
        double t1, phi1, x1, ...;
        volatile double t2, phi2, x2 ...;
        t1 = sqrt(drand48());
        t2 = sqrt(drand48());
        if (t1 != t2) {
            // error is detected;
            errors++;
            if (errors > THRESHOLD) {
                disable the core and exit;
            } else {
                do correction;
            }
        } else {
            Store the correct result;
        }
    }
    phi1 = 2.0 * M_PI * drand48();
    phi2 = 2.0 * M_PI * drand48();
    if (phi1 != phi2) {
        error is detected;
        Disable the core or correct the error;
    }
} 
// the remaining code is not listed

Figure 2.3: Scalar mode error detection for the loop in Aobench. For the sake of brevity, the duplication of other variables is not shown in this figure.
Vector mode error detection

This section presents the vector mode error detection technique. Figure 2.4 illustrates how the instructions in Figure 2.2 are duplicated in vector mode to detect errors. For the sake of brevity, the duplication of only one instruction is shown in this example. For example, in order to detect the errors for the \texttt{sqrt} operation, we need to check the equality of the high 128 bits and the low 128 bits for the result register \( t \) after the operation is performed on the vector register \( rnd \).

Our technique is applicable to both single bit and multibit soft errors. It only cannot detect the error when the same bits of the two results are flipped simultaneously, but the possibility of this extreme case is negligible. Actually, no existing instruction duplication strategy is able to handle this case. The performance and energy overhead as compared to duplicating scalar instructions will be largely or even completely offset by vectorization since multiple datum are loaded and executed in parallel. However, there will likely be a loss of performance compared to vectorization using full vector register width. For instance, only two duplicated f.p. operations are performed by an AVX f.p. instruction as compared to four for non-duplicated code.

An f.p. arithmetic instruction is vectorized first and then duplicated to pack a XMM or YMM register (for SSE or AVX instructions, respectively). The results in the output register are compared for equality. If the related two values are identical (i.e., the value obtained from \( \text{mm256\_cmp\_pd}() \) for AVX or \( \text{mm\_cmpneq\_pd}() \) for SSE is a zero vector), no failure is present. Otherwise, similar to the case for scalar mode error detection, an error is detected and the error counter is incremented by 1.

Figure 2.5 shows how this is accomplished. First, array elements \( B[i], B[i+1] \) and \( C[i], C[i+1] \) are loaded into YMM1 and YMM2 vector registers, respectively. An AVX instruction is then used to replicate the data in the remaining two elements of each YMM register.
for (i = 0; i < nphi; i += 2) {
    rndLow = _mm_load_pd(&thRand[i]);
    rnd = _mm256_broadcast_pd(&rndLow);
    t = _mm256_sqrt_pd(rnd);
    t_d = _mm256_permute2f128_pd(t, t, 1);
    d = _mm256_cmp_pd(t, t_d, _CMP_NEQ_OQ);
    if (_mm256_movemask_pd(d) == 0xf)
        // error is detected
        errors ++; // increment errors
        if (errors > THRESHOLD)
            // disable the core and exit;
        else {
            do correction;
        }
    } else {
        Store the correct result;
    }
    // Perform error detection/correction for the rest code;
}

Figure 2.4: Error detection and correction for the loop in the Aobench.

Figure 2.5: Vector mode duplication for error detection. YMM registers are used by AVX instructions. All YMM registers in this figure are 256 bits, e.g. ymm3 and ymm4 contain 4 double f.p. values.
of the two YMM registers is performed next and the result is written into YMM3. The values in register YMM3 are permuted and written into register YMM4 because the fields of a vector register are not accessible directly for comparison. A vector mode comparison is performed on YMM3 and YMM4. A fault is detected if the results are not equal and a recovery can be performed by recomputing a value in error for the 3rd time and detecting identical values. This works because soft errors are short-lived.

Figure 2.6 shows the X86 assembly of Line 3-7 in Figure 2.4. Two observations can be made as to why the vector mode duplication should not severely worsen the performance. First, the cost of error detection through vector mode duplication is only 3 extra vector instructions: permutation, comparison, and masking, which are not very expensive on modern processors. A vector comparison only requires 3 cycles and the vector masking and permutation consume 2 cycles each on the Sandy Bridge and Ivy Bridge processors. Second, out-of-order instruction issue and ILP capabilities of a modern processor will likely partially overlap such instructions with others.

```
1  vbroadcastf128 (%rsi,%r14,1),%ymm0
2  vsqrtpd %ymm0,%ymm9
3  vperm2f128 $0x1,%ymm9,%ymm9,%ymm1
4  vcmpneq_oqpd %ymm1,%ymm9,%ymm2
5  vmovmskpdp %ymm2,%eax
```

Figure 2.6: Part of AVX assembly code from Aobench.

### 2.2 Experimental Evaluation

This section presents and analyzes experimental results measuring the impact of our technique. The implementation of vectorization and error detection was performed on C/C++ benchmarks. ICC-14.0.3 compiler was used, with “-O3 -xsse4.2/-xavx” options to compile a set of benchmarks. All the experiments are run on a 8-core Sandy Bridge Core i7 processor with configuration listed in Table 2.1.
Table 2.1: System configurations

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Intel(R) Core(TM) i7-2600</td>
</tr>
<tr>
<td>Frequency</td>
<td>3.4GHz</td>
</tr>
<tr>
<td>L1 I/D cache</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256KB</td>
</tr>
<tr>
<td>L3 cache</td>
<td>8MB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Ubuntu 11.10 Linux 3.0.0</td>
</tr>
</tbody>
</table>

The benchmarks used were: IRSmk and Crystalmk from LLNL [1], Aobench [5], FFT from StreamIt benchmark suit [33], Blackscholes from PARSEC 3.0, Spectral-norm and Mandelbrot from benchmarksgame, and 183.equate from SPEC2000. One manual optimization performed for Spectral-norm benchmark was replacing division in the \( \text{sum} += b/a \) statement in Spectral-norm benchmark with reciprocal and multiplication instructions. This version offers much better performance since vector reciprocal only takes 5 cycles but vector division needs 10-14 cycles.

Table 2.2: Profiling information of each benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>lang.</th>
<th>LOC</th>
<th>loops</th>
<th>hot</th>
<th>exe.</th>
<th>T%</th>
<th>vec%</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRSmk</td>
<td>C</td>
<td>327</td>
<td>7</td>
<td>3</td>
<td>99</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Crystalmk</td>
<td>C</td>
<td>301</td>
<td>31</td>
<td>5</td>
<td>82</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Aobench</td>
<td>C</td>
<td>266</td>
<td>6</td>
<td>2</td>
<td>98</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>C</td>
<td>260</td>
<td>13</td>
<td>3</td>
<td>59</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Spectral</td>
<td>C</td>
<td>121</td>
<td>6</td>
<td>4</td>
<td>99</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>C++</td>
<td>128</td>
<td>6</td>
<td>3</td>
<td>69</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>Blackscholes</td>
<td>C</td>
<td>393</td>
<td>5</td>
<td>2</td>
<td>94</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>Equake</td>
<td>C</td>
<td>1513</td>
<td>88</td>
<td>3</td>
<td>74</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

The impact of our approach is measured using hardware performance counters in benchmark execution. Each benchmark was compiled in several ways to gauge the effect. The different versions used were: the original program (base), the original program with each f.p. instruction duplicated (b_dup) in hot loops, base using SSE intrinsics in hot loops (sse), base using instruction duplication with SSE (s_dup), base using AVX intrinsics in hot loops (avx), and
base using instruction duplication with AVX (x.dup).

Table 2.3: The execution time (s) and energy consumption (J) of baseline and baseline with duplication.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>base T</th>
<th>b.dup T</th>
<th>base E</th>
<th>b.dup E</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRSmk</td>
<td>29.36</td>
<td>60.46</td>
<td>632.67</td>
<td>1263.51</td>
</tr>
<tr>
<td>Crystalmk</td>
<td>21.83</td>
<td>52.88</td>
<td>494.15</td>
<td>1162.84</td>
</tr>
<tr>
<td>Aobench</td>
<td>33.32</td>
<td>73.22</td>
<td>670.67</td>
<td>1501.51</td>
</tr>
<tr>
<td>FFT</td>
<td>3.64</td>
<td>6.9</td>
<td>209</td>
<td>283.53</td>
</tr>
<tr>
<td>Spectral</td>
<td>13.02</td>
<td>26.03</td>
<td>236.19</td>
<td>471.2</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>12.8</td>
<td>34.31</td>
<td>236.28</td>
<td>705.43</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>1.35</td>
<td>3.04</td>
<td>31.96</td>
<td>73.76</td>
</tr>
<tr>
<td>Equake</td>
<td>16.65</td>
<td>33.68</td>
<td>389.54</td>
<td>728.1</td>
</tr>
</tbody>
</table>

The results of profiling baseline (base) and baseline with duplication (b.dup) are shown in Table 2.2. It shows the lines of code, the number of hot loops to which our technique was applied, fraction of execution time in the hot loops (exe. T%), and fraction of vectorizable code (vec%) in hot loops (e.g., the percentage of the code that could be replaced with SIMD intrinsics in the hot functions). Table 2.3 details the execution time (T) and the energy consumption (E) of baseline and baseline with duplication.

2.2.1 Evaluation

This section presents comparison of performance and energy consumption of different versions of each benchmark obtained using hardware performance counters. All results are relative.

Performance loss of SSE instruction duplication over baseline. Now we compare the performance of instruction duplication with SSE to the baseline without duplication. The execution of instruction duplication with SSE is similar to the scalar mode because we use one slot in the SSE vector for the original instruction and the other one is used for duplication. This replaces the original f.p. scalar instruction with a single SSE instruction,
Figure 2.7: Slowdown of instruction duplication for: (a) SSE intrinsics over the baseline, (b) scalar and vector instruction duplication.
which is only slightly slower. Figure 2.7(a) shows the average performance slowdown is only 1.2x due to two major reasons. First, comparisons are introduced to check the errors. Second, we have to broadcast the data to both elements of the vector register after it is loaded. This is a low overhead to implement error detection.

**Impact of instruction duplication.** Figure 2.7(b) shows the impact of using scalar mode instruction duplication, vector mode instruction duplication using SSE, and vector mode instruction duplication using AVX. Each version is compared to the corresponding version without duplication.

The execution time at least doubles for the baseline when using scalar mode instruction duplication (except FFT). This is reasonable because much more instructions (e.g., duplicates, checking code) are executed in the hot loops for error detection in scalar mode. For the memory intensive benchmarks like FFT, the slowdowns are slightly less than 2x due to the following reason. Although all FP instructions are duplicated and comparison and masking instructions are applied, the performance overhead caused by additional instructions is still hardly noticeable. However, the performance slowdown is higher for the compute-intensive benchmarks (e.g., Mandelbrot) since more instructions are executed for error detection. One can argue that this is a reasonably fair price for soft error detection.

The SSE and AVX implementations have an average slowdown from vector mode instruction duplication of 1.54x and 1.55x, respectively. It is also always noticeably less than 2x for AVX. Even for SSE with duplication all benchmarks have slowdown of less than 2x. This is due to at least two reasons. First, the compiler can still help to optimize code after our hand vectorization and duplication. Second, comparison instructions in SSE and AVX versions are performed in vector mode.

IRSmk and FFT benchmarks only slow down by 4% and 14%, respectively, from instruction duplication with AVX intrinsics. There are two reasons for this. I) The statements in their
hot loops can be fully vectorized by our hand vectorization, i.e., we can replace all the scalar instructions with the SSE/AVX alternatives. II) The two benchmarks are memory-intensive: the instruction mix of IRSmk is 47% loads/stores and 24% FP operations; the FFT contains 52% memory operations and 17% FP instructions. Thus, the increased time to execute duplicated FP instructions remains a small portion of the overall execution.

**Cost of error detection.** One can argue that some form of software error detection has to be used in future unreliable processors, especially so if they are used to build Exascale or other large systems. Let us now compare the performance of the proposed error detection scheme using vector instructions with the scalar duplicated instruction version (base_dup).

Figure 2.8 shows that vector mode instruction duplication achieves a significant speedup compared to the baseline with scalar duplication, e.g., 1.78x and 2.73x for SSE and AVX, respectively. This is true for both SSE and AVX versions of every benchmark and on average. This also shows that one can always achieve software-based fault detection but with different effect on performance.
2.2.2 Energy consumption

Let us now compare the energy consumption of the vectorization and error detection techniques on the Intel Sandy Bridge Core I7 processor (measured using Likwid 3.1.1 [103]). The energy information was collected from the PKG counter while disabling all other cores to avoid their interference.

Energy reduction from vectorization. Figure 2.9 shows the fraction of energy reduction due to vectorization using SSE and AVX intrinsics compared to the baseline. The use of SSE and AVX intrinsics reduces the average energy consumption by 20% and 34%, respectively. The energy savings are in part due to the reduced execution time.

The AVX instructions usually consume more power than scalar and SSE instructions because additional voltage will be applied to the core when these instructions are detected (i.e., the higher voltage lasts for 1ms after the finish of the last Intel avx instructions) [39]. This can make the processor hotter. In such a case the processor needs to scale down frequency to maintain thermal design power. Even though AVX instructions require more power,
Figure 2.10: Energy increase of instruction duplication.

Figure 2.11: Energy reduction of error detection using vector mode instruction duplication.
they have higher energy efficiency since much time is saved from better parallelism. In addition, vectorization effectively reduces instruction count and pipeline/I-cache pressure further reducing energy consumption.

**Energy increase from instruction duplication.** Figure 2.10 shows the normalized energy consumption of instruction duplication in both scalar mode and vector mode relative to the versions without duplication. The average energy increase of the baseline with instruction duplication is 51%.

Vector mode instruction duplication only requires 35% and 29% more energy compared to the code with SSE and AVX intrinsics but without duplication. Again, the implementation with AVX achieves higher energy efficiency than the scalar and SSE implementations due to better parallelism.

**Energy reduction of vector error detection.** Let us compare the relative energy of scalar-mode vs vector mode duplication. Figure 2.11 shows the energy reduction of the proposed error detection technique using vector instructions over energy consumed by the scalar baseline with instruction duplication. The energy reduction from vector mode instruction duplication is significant. The vector mode instruction duplication for SSE and AVX has, on average, 40% and 53% reduction, respectively, over the baseline with instruction duplication. Overall, the energy reduction using AVX instructions is still much higher than when SSE instructions. This is because AVX offers twice as much execution bandwidth as SSE, therefore yielding much better energy efficiency.

### 2.3 Summary

This chapter proposed a framework to detect soft errors in floating point instructions of vector mode instruction duplication relying on using intrinsic functions. No hardware replication
is required. This saves significant chip area and energy and eliminates any potential impact on cycle time. Performance and energy using SSE and AVX duplication were measured and showed that our approach can effectively implement the error detection. For instance, our hand vectorization on a variety of benchmarks showed that, compared to error detection through scalar instruction duplication, vector mode redundant execution achieves 1.78x and 2.73x average speedup for SSE and AVX instructions, respectively. The proposed vector-mode soft error detection has two other potential benefits.

First, the technique detects soft errors in f.p. operations at run-time. This is likely less expensive than checkpoint-restart approach that saves a large state.

Second, the proposed error detection technique can be used in conjunction with energy reduction techniques. For instance, FPUs can use DVFS even though lower voltage increases soft error rates. This is because the errors will be detected and corrected by software anyway.

Therefore, this result motivates us to modify a compiler to generate code for vector error detection automatically for all arithmetic operations and even memory instructions. In addition, a common feature of the benchmarks employed in this chapter is that they all have “hot” loops. Moreover, the vectorization and instruction duplication will be even less tedious if these statements have friendly data access patterns, i.e., no structure to array (SoA) or complicated functions are involved. Nonetheless, more comprehensive approaches should be devised in the future to cover more additional applications, including those without hot loops or with complicated hot loops.
Chapter 3

Compilation Flow

We’ve described the manual method to vectorize the original instruction and its duplicate for error detection in Chapter 2. Relatively low performance and energy consumption have been also measured compared to the scalar duplication due to under-utilization of vector units and notably fewer extra instructions. The manual vectorization provided us a feasibility study of vector-based instruction duplication for error detection. While effective in reducing performance and energy overhead, this approach is subject to several inherent limitations, which makes it only confine to a limited number of applications.

- **Flexibility.** One could argue that the hand vectorization would have slightly higher overhead than the automatically generated code since it might lose some compiler optimization opportunities through using intrinsics.

- **Scalability.** Even though the it could work for small benchmarks, it entails a lot of effort to rewrite large benchmarks, such as SPEC suites. For the small benchmarks, we can only focus on the “hot” loops as they count of a majority of execution time. Even replacing the whole benchmark with intrinsic functions may only require reasonable effort. However, some large applications may not even have “hot” loops or may
have a variety of “hot” loops. Rewriting these applications using intrinsics is quite demanding.

- **Portability.** Intrinsic functions offer programmers an easier way to vectorize code, but it also suffers from portability issues as the intrinsic instructions are fundamentally different for various architectures; that is, code writers have to prepare multiple versions of intrinsic implementations for the same application to support different architectures. This significantly increases the programming burden if the error detection is required since the programmers must have expertise in a variety of intrinsics.

- **Readability.** The code becomes extremely unreadable after hand vectorization using intrinsics and insertion of error checking code as it is inevitably marred by verbosity.

Therefore, a more advanced approach that involves much less or virtually no manual modification at the source level is of significance. One solution is to add a compiler pass to achieve instruction duplication so that error detection is transparent to programmers and it could be enabled conveniently only when it is necessary.

This chapter describes our SIMD-based compilation approach to instruction duplication using the LLVM compiler infrastructure [48]. To be specific, it assumes Intel SIMD instructions and uses both SSE and AVX2 instructions. Of course, the approach can be applied to other SIMD instruction sets. Examples below use floating point operations, but the same approach is also used for integers.
Figure 3.1: Scalar (a) and SIMD-based (b,c) source code duplication and error checking. (b): codes without SIMD instructions and (c): codes with SSE instructions.
3.1 Compiler Implementation

3.1.1 Arithmetic Instructions, Logical Instructions, and Bit-wise Instructions

Figure 3.1 illustrates the basic idea of scalar and SIMD-based instruction duplication. The figure uses a source-level “instruction” for simplicity: \( A[i] = B[i] + C[i] \). For illustrative purpose, only add operation is given. But other arithmetic operations, logical instructions and bit-wise operations would be also processed similarly. Figure 3.1(a) shows the scalar instruction duplication that is commonly used by the prior art. It assumes that arrays \( A, B, \) and \( C \) are in memory. Two independent instructions are generated, \( A[i] = B[i] + C[i] \) and \( A'[i] = B[i] + C[i] \) using the same input operands loaded from memory. A comparison is inserted to check that \( A[i] \) and \( A'[i] \) are equal. A branch is also required to deal with an error. A recovery can be initiated by re-executing the same instruction a third time and performing majority voting, or by using software checkpoints.

Figure 3.1(b) and Figure 3.1(c) demonstrate SIMD-based instruction duplication and error checking. Two possible cases are illustrated: code in Figure 3.1(b) has no SIMD instructions and code in Figure 3.1(c) has SSE instructions. There are two ways to duplicate memory instructions. Here we illustrate duplicating load instructions using broadcast primitives. This way can protect the data that is being used. We will present a way to fully protect memory instructions in Section 4.5. In the former case, the operands \( B[i] \) and \( C[i] \) are loaded into the low quadword of two XMM registers. A broadcast instruction duplicates the value in low quadword to the high quadword (the dark grey boxes in the second row). An SSE addition is performed using the SIMD registers with duplicated operands. The results, \( A[i] \) and \( A'[i] \), are in the same XMM register after the computation. One only needs to check the equality of the words in the same register for error detection. However, the SSE (and AVX)
instruction set does not contain an instruction for such a comparison. The result register thus needs to be copied with a shuffle to another register before the comparison.

Figure 3.1(c) shows the duplication in a code with SSE instructions (vector length=2). Duplication in this case requires a wider register, e.g. a YMM register (vector length=4). Otherwise the code is the same as the one shown in Figure 3.1(b).

Finally, a code containing SIMD instructions using the widest SIMD registers available on the target processor, e.g., a YMM register for AVX instructions in our case, cannot be duplicated in this manner. The scalar duplication approach in Figure 3.1(a) or other approaches (will be discussed later) could be used in this case.

3.1.2 Memory Instructions

As shown in Figure 3.1(a), the scalar instruction duplication has to loads the value from the same memory location twice to guarantee two separate execution path have independent execution. Another load duplication method commonly used by scalar instruction duplication techniques, e.g. [87], is to load the value from memory once, and then adding a move instruction to copy the loaded value into a redundant register. Both of these two methods are effective in detecting errors that happen on the arithmetic units as the equality of the original instruction and its replicate will be checked at certain points. However, this scalar load is not effective at least for the following two cases.

First, once there was an error in the memory to be read because two execution path will both have the erroneous data. Additionally, once loaded, the erroneous data may be reused by the processor since it is likely to be held by the cache. Therefore, all the subsequent scalar loads starting from this point will have the same faulty data in the data cache. That said, all the instructions that depend on the load will also use the corrupted data but couldn’t
be detected as the compiler is not aware of the run-time information of the cache. The side effect from cache could be solved by disabling the cache hierarchy altogether, but it will inevitably lead to a substantial performance penalty. Another alternative method is to flush the cache line for the duplicated load whenever the it needs to read from memory. This method was used by [105] to protect against the timing attacks. Extra instructions have to be inserted before the duplicated instruction to achieve the invalidation of cache for the duplicate load.

Second, researchers in [114] have observed that the original load and its duplicate can be present at two adjacent pipeline stages at the same cycle, i.e., one is at the register access stage and the other is at the execute stage, where only a single fault injected might corrupt these two loads simultaneously. Duplicating memory using broadcast instructions can initiate the data duplication. But it suffers the same drawbacks as scalar duplication. In order to guarantee that duplication can both initiate two data points for execution (one is the original data and the other is the duplicate), our vector instruction duplication approach loads two data from two different memory locations. Gather instructions can effectively eliminate the extra load used in the previous scalar instruction duplication. Duplication in this manner avoids executing identical loads sequentially since it doesn’t attempt to take the spatial redundancy. The vector duplication scheme thus makes the countermeasure resistant to the above single fault injection.

Not only memory addresses can be protected, memory contents can also be hardened. For instance, we can replicate the data (e.g. with size $S$) stored at memory address $M$ on its continuous memory starting from $M+S$. Instead of loading the value $A$ at address $M+offset$, we gather a pair of values $A$ from address $M+offset$ and $A'$ from address $M+offset+S$ and then pack them into a SIMD register as shown in Figure 3.2(a). A comparison is then performed to validate the equivalence of these two values to check if the loaded memory content is corrupted.
Similar operations can be performed to protect stores as well using **scatter** instructions as represented by the right side of Figure 3.2. However, we would have to read back the two values stored by a scatter instruction to testify if there was a fault injected to the content at one of the particular memory addresses, which requires another gather instruction and one more comparison. These procedures enable countermeasure on loads and stores at the cost of high performance overhead because gathers and scatters normally require much more \( \mu \)ops than a **mov** instruction, i.e., a gather is completed by up to 6 \( \mu \)ops depending on the number of used elements but a **mov** instruction only needs 1 \( \mu \)ops on the Knights Landing architecture [28].

![Memory instruction duplication using (a) gather, (b) scatter.](image)

**Figure 3.2:** Memory instruction duplication using (a) gather, (b) scatter.

### 3.1.3 Branch Instructions

Before each branch instruction, we will check the correctness of the branch condition because it has been vectorized from the previous instructions. Even though branch conditions are verified, there is still a chance that a branch gets incorrectly directed without being detected.
For instance, an error may happen during the execution of the branch or the instruction pointer might be corrupted therefore its update will contributes to a wrong target. Our compiler currently does not implement branch target checking, unlike the extensions to EDDI [73] and SWIFT.

**Branch Protection.** Errors occur in the control flow conditions are covered by duplicating and checking the branch condition. However, errors happen in the control flow target, e.g. faults in the destination of a branch, go undetected by our approach since the destination is not duplicated. While proposing novel schemes to protect branch targets is out of the focus on this dissertation, some possible software approaches, e.g. [96, 16, 72, 87, 31, 106, 43], could be employed to achieve this goal.

### 3.1.4 Call Instructions

There are two major types of call instructions, namely a function call and a library call.

Function calls are not duplicated since there is no equivalent vector form available for an arbitrary function, particularly for the user defined ones. However, function calls may affect the outcome of a program since parameter corruption will very likely lead to an incorrect output. Instead of duplicating a function call, which is unrealistic, we treated function calls as points for error checking where each parameter that is passed by value will be verified before the call is taking place. Faults will be detected once there is a mismatch between the original and a duplicate of a parameter. In the callee, a parameter will be packed into a vector register before the first time it is used. This is accomplished by using the *broadcast* instruction to make a vector form as it had in the original caller function so that the other dependent instructions will view it as a vector operand. On return, the returned value in the vector form will be checked. Once the two values in a vector register are identical, only one of them will be returned.
Packing and unpacking the parameters adds performance cost and it may also introduce vulnerability concerns. To mitigate these two issues, one possible solution is to modify the function by changing the return type and each of the parameter type from scalar to vector. With this modification, the parameters would keep the value in the same vector register as they were in the caller without the need to pack and unpack them. In addition, it is unnecessary to verify the values in the vector registers before return instructions since they will be taken by the callers, and they will be finally checked at other synchronization points, e.g. stores and branches. However, this would require more complicated program analysis since we have to look at all the calls and call sites in a program. Therefore, our current implementation only relies on using packing and unpacking operations for function calls. It leaves modification of function signatures as a future work.

Most of the mathematical library calls are duplicated since there are vector versions of them available. For instance, Intel Math Kernel Library (MKL) and Short Vector Math Library (SVML) support vectorized forms of many library calls, e.g. trigonometric functions, logarithmic functions, and sqrt, etc. One can vectorize these functions conveniently according to the input requirement of the LLVM front-end and leave the selection of SIMD intrinsic function calls to the back-end. Most of other library calls, such as systems calls, are not duplicated. They are hence handled the same way as function calls.

### 3.1.5 Other Instructions

While almost all the instructions discussed above are protected, some instructions are fundamentally difficult to harden at the IR level.

- A portion of mov instructions. Compilers generate mov instructions to spill variables from register to memory at the register allocation stage when there are not enough registers. These instructions are invisible at the IR stage, therefore couldn’t be dupli-
icated. The best practice to guard these instructions is probably through modification of either the register allocation pass or the assembly.

- Stack manipulation instructions, e.g. `push` and `pop`. Compilers generate push and pop instructions to manage the stack, e.g. handling function call parameters, these instructions are added at the code generation stage. We cannot prevent these instructions from corruption, despite the fact that the value of parameters are duplicated at the beginning of a call and verified before the call.

- Complex instructions for specific algorithms, e.g. cryptographic instructions (aesenc and aesdec, etc.). These instructions could be protected by scalar duplication, e.g. one could run these complex instructions twice and compare the outputs since the vector counterparts of them are unavailable.

However, these instructions are quite rare in a program. One can argue that it is safe to ignore them for duplication since we are not targeting 100% error coverage but trading off coverage vs performance and energy overheads.

### 3.1.6 Multibit Errors

The error injection experiments revealed that the proposed SIMD based error detection approach is sufficient to detect single-bit error in almost all but a few corner cases. Nonetheless, it is still debatable in case of multibit faults. Multibit errors can go undetected in two possible cases: I) when the same bit is flipped at both the high and low quadword (128-bit) of a xmm (ymm) register during computation, II) when a bit is flipped in a vector register with redundant data and the comparison is also flipped so that it would not branch to error handling code. As analyzed in the dissertation, these types of multibit faults are rare enough to be safely ignored.
Next, an overview of the compilation framework is given in Section 3.2. Then, the use of conventional and our SIMD-based techniques to replicate instructions is presented in Section 3.2.1. Finally, different ways of inserting error checking code are discussed in Section 3.3.

### 3.2 Overall Compilation Process

#### 3.2.1 Instruction Duplication

Figure 3.3 shows the framework of the proposed solution using the LLVM compiler infrastructure [48]. The input to the framework is application source code in C/C++ \(^{1}\). The LLVM front-end compiles the source code and converts it into the LLVM intermediate representation (IR). A large set of standard transformations and optimizations are applied to the IR, producing an optimized LLVM IR form.

Our module, the grey box in Figure 3.3, starts with the optimized IR as the input and performs duplication in the IR code in a new LLVM pass. The LLVM IR uses a language-independent instruction set represented in a static single assignment (SSA) form. The new instruction duplication pass visits each individual IR instruction and analyzes it to determine

\(^{1}\)Or any other language that is supported by LLVM.
```c
//code before this point is omitted
for (i = 0; i < nodes; i++) {
    Anext = Aindex[i];
    Alast = Aindex[i + 1];

    sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] + A[Anext][0][2]*v[i][2];
    sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][1][2]*v[i][2];
    sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] + A[Anext][2][2]*v[i][2];

    Anext ++;
    ...
}
//code after this point is omitted
```

Figure 3.4: Instruction duplication example. (a) Source code, (b) Partial LLVM bitcode.
Figure 3.5: Transformed LLVM bitcode for the LLVM bitcode in Figure 3.4(b). A highlighted IR instruction is a duplicate, followed by the original instruction.
if it is necessary to duplicate and vectorize this instruction (recall that branches/jumps are not duplicated). The output of the pass is a modified IR form where an original instruction and its replica are “packed” in the same SIMD instruction. More details about the transformation will be given in the next section. A second new pass then is used to remove the original scalar instructions for cleanup purpose. Finally, another pass adds the checking code for all duplicated instructions to compare results and branch to the error handler once an error is detected.

The transformed IR is the input to the (unmodified) LLVM back-end which performs register allocation, additional optimizations, and code generation. The code generated in this work is for Intel processors with the AVX2 instruction set and can thus deal with both integer and floating point instructions. The proposed technique is implemented in the single threaded context, but we believe it can be easily extended to multithreaded codes. Also, the same idea is applicable to other platforms that supports

Let us recap how different instruction classes are dealt with for soft error detection in this work.

**ALU instructions.** These instructions are duplicated via an equivalent SIMD instruction and their data is placed into SIMD registers.

**Memory instructions.** Loads and stores are protected using gather and scatter primitives, respectively.

**Branches.** The branch condition computation is duplicated and checked. The PC update is not visible to user code and thus cannot be protected by instruction duplication. ²

**Function Calls.** Function calls are not duplicated. Some library functions can be protected by duplication.

²Existing software solutions based on run-time signature or assertions [16, 72, 87, 31, 106] can be utilized, these are orthogonal to our work.
The scalar and the SIMD-based instruction duplication at the IR level are discussed next. An example, shown in Figure 3.4(a), will be used to illustrate the ideas. Figure 3.4(a) shows a code snippet from the smvp function of 183.equake benchmark. Figure 3.4(b) shows (partial) IR code produced by the LLVM front-end. The IR code contains both scalar and vector operations. The latter can be distinguished by “\(2 \times \text{double}\)” type. In fact, LLVM doesn’t perform loop level vectorization for this code as the loop index \%Anext.1 is incremented only by 1 in each iteration (line 18 in 3.4(b)). However, Lit vectorizes some codes inside this loop since three continuous array elements are loaded to compute \(\text{sum}\,0\), \(\text{sum}\,1\), and \(\text{sum}\,2\), respectively.

### 3.2.2 Scalar Instruction Duplication

The scalar (or standard) instruction duplication accesses each IR instruction and generates a duplicate instruction. This is the approach used in EDDI and SWIFT which only considered scalar instructions. One major difference in our case is that a program may contain both scalar and vector instructions prior to duplication. Thus, even when we perform scalar duplication, the compiler duplicates any existing vector instructions.

Figure 3.5 shows the IR code after the standard duplication is applied. The highlighted instructions (with underscore added to the output register) in the figure are the duplicates followed by the original instructions. For example, the multiplication and addition instructions, i.e., \%mul69, \%add70, \%Anext.1, are duplicated as \%mul69, \%add70, \%Anext.1, respectively. The vector instructions, such as \%17, \%23, and \%24, are handled the same way as the scalar instructions. Only instruction duplication is shown, the error checking code is inserted in another pass.
for.body: ; preds = %while.e, %ent

%Phi = phi <2 x i64> [%iv.nxt1, %while.e], [ zeroinitializer, %ent]
...
%4 = load double* %3, align 8, !tbaa !16
%ld4 = insertelement <2 x double> undef, double %4, i32 0
%ld4_ = shufflevector <2 x double> %ld4, <2 x double> undef, <2 x i32>
zeroinitializer
...
%13 = insertelement <2 x double> undef, double %4, i32 0
%13_ = shufflevector <2 x double> %ld4_, <2 x double> %ld4_, <4 x i32>
    <i32 0, i32 2, i32 1, i32 3>
...
%17_ = fmul <4 x double> %14_, %16_
...
%23_ = fmul <4 x double> %20_, %22_
%24_ = fadd <4 x double> %17_, %23_
...
%arridx65 = getelementptr double* %32, i64 1
%34 = load double* %arridx65, align 8
%ld34 = insertelement <2 x double> undef, double %34, i32 0
%ld34_ = shufflevector <2 x double> %ld34, <2 x double> undef, <2 x i32>
zeroinitializer
%mul69_ = fmul <2 x double> %8_, %ld34_
%add70_ = fadd <2 x double> %mul61_, %mul69_
...
%Anext.1_ = add <2 x i32> %0_, <i32 1, i32 1>
%cmp1_ = icmp slt <2 x i32> %Anext.1_, %1_.
%zext1_ = sext <2 x i1> %cmp1_ to <2 x i64>
%bc1_ = bitcast <2 x i64> %zext1_ to i128
%msk1_ = icmp ne i128 %bc_, 0
br i1 %msk1_ label %while.b, label %while.e

Figure 3.6: LLVM IR after SIMD-based instruction duplication.
3.2.3 SIMD-based Duplication

The SIMD-based duplication of an instruction in the IR is based on the instruction’s class and data type. In all cases, the original instruction is retained for the duration of this pass to simplify compilation and is removed in a separate pass.

1. **Scalar integer instructions use scalar registers.** For example, instruction `%A_{next},1` in Figure 3.4(b) will be allocated to a 32-bit scalar register. A scalar integer instruction will be changed to an SIMD instruction. For instance, Figure 3.6 shows that `%A_{next},1` is replaced by its vector counterpart, `%A_{next},1_`.

2. **Scalar floating point instructions.** Intel processors use SIMD registers for floating point operations by default, thus duplication of a scalar floating point instruction only requires value duplication. For example, `%mul69_` and `%add70_` in Figure 3.6 will use the same registers as used by `%mul69` and `%add70` in Figure 3.4(b), respectively. The technique used for these instructions corresponds to Figure 3.1(b). The solution for scalar integer instructions discussed above can be applied if the floating point operations do not use SIMD registers by default.

3. **SSE instructions.** Duplicating SSE instructions, such as `%17`, `%23`, and `%24` in Figure 3.4(b) requires wider registers, e.g. AVX2 registers. For example, instructions `%17_`, `%23_`, and `%24_` in Figure 3.6 correspond to the case in Figure 3.5.

4. **AVX instructions.** There are two possible solutions for this case: a) perform AVX instruction duplication as in the scalar approach (shown in Figure 3.1(a), and b) generate only SSE instructions and duplicate them using AVX instructions. We used the later solution in this work.

5. **Memory instructions.** We protect all load and store addresses by using gather and scatter instructions to duplicate memory access computation.
Recall that currently we do not duplicate loads (mov) instruction. However, the compiler generates “vmov...” instructions (such as vmovhlps and vmovddup, etc) to load SIMD registers, e.g. insertelement and shufflevector instructions in the IR shown in Figure 3.6. A value is loaded once into an SIMD register and then broadcast to an unused lane in the register, e.g. %4_ in Figure 3.6.

A \(2 \times 1b\) result of a vector comparison needs to be converted to an SSE form. Otherwise, LLVM has to perform unpacking and packing for this comparison, the same way as it handles instructions that cannot run in the SIMD mode – such as integer division (see below). This requires sign extending the result to a \(2 \times 64b\) vector instruction first. A bitcast IR instruction is then used to make the value compatible to an SIMD register (e.g. 128 bits for SSE). The comparison instruction, such as %cmp1_ in Figure 3.6 is an example of this. LLVM then generates vector comparison and branch instructions, e.g. vpcmpeqq and vptest, in the back-end.

Most of the instructions in the original code become unnecessary after instruction duplication is completed. For example, most of the ALU instructions in Figure 3.4(b) can be removed because each of them has an SIMD counterpart in the modified IR code. However, some instructions do not have an SIMD equivalent on Intel processors, e.g. integer division and int32 to double conversion. In this case LLVM unpacks the two values from an SIMD register and then creates two scalar instructions for each value. Finally, the values produced by these two scalar instructions are packed into a vector register. The packing and unpacking operations would require several instructions. Fortunately, these instructions only accounts for a very limited portion in most applications. They, thus, don’t severely impact the code size, performance, and energy consumption.
Figure 3.7: (a) A basic block with 4 instructions. (b) The basic block after SIMD-based instruction duplication. (c) Error checking after each duplicated instruction, (d) Error checking at only before a store.
3.3 Error Checking

Error checking is performed in a separate LLVM pass, after instructions are duplicated and the unneeded instructions are removed. Each checker is essentially composed by at least three instructions: a shuffle, a comparison, and a branch to an error handler. Checking can be performed in two different ways, both of which are implemented and compared.

- Check after each duplicated instruction. The advantage of checking at this granularity is that error correction can be initiated immediately. The disadvantage is that it incurs a very high overhead.

- Check only at certain program points, such as before stores, function calls, conditional branches, etc. Overheads are thus significantly reduced. However, recovery becomes harder as instructions may be committed and register state may have changed by the time the error is detected. This probably works best with checkpointing and the static analysis technique proposed in [23].

The way checking is performed in the second case depends on the program point it is inserted at. But the main idea is to only check store values before they can change program state. All other operations just propagate values to the stores/sink points.

- Stores. The duplicated store operand (aka. the value to be stored to memory) is checked.

- Conditional Branches. The condition evaluation result is checked.

- Calls. It is assumed that parameters are pushed onto the stack before the call. Thus they are stored to memory and will be checked. A parameter passed in a register needs to be checked separately. Parameters that are passed by reference are not duplicated
because stack memory accesses are assumed to be reliable. This approach was also used in prior research, e.g. EDDI and SWIFT.

Figure 3.7 illustrates the two different error checking techniques. Figure 3.7(a) shows a code block with four instructions, only the last of which is a store. Figure 3.7(b) is the code block after our SIMD-based duplication. The checkers inserted after each instruction are shown in Figure 3.7(c). Figure 3.7(d) shows error checking only before a store, where the store operand is validated before being written to memory. Only one checker is needed in this case.

While inserting checkers only at the certain program points could significantly reduce the performance overhead. It still suffers some penalty in performance and brings more points of failure as more instructions are required. For example, we may still need add a checker at the end of many basic blocks and each checker would need several steps, e.g. permutation, comparison, and branching, etc. These steps is essentially composed of several instructions. One possible way to further reduce the overhead is the design of a special instruction that can compares the most significant and least significant 64 bits (128 bits) of a xmm (ymm) register directly. Another applicable manner could be the implementation of a piece of special hardware (e.g. a compactor) to check the integrity of a SIMD register.

The following two scenarios can happen due to the presence of the bit flip in the added error checking code.

(a) The check result is evaluated to be true. In this case, the program will branches to the error handling code to process the caught error, yet this is false positive for the original program.

(b) The check result is evaluated to be false. That being said, an error had already occurred, but another error is happening again in the error checking code. As the checkers
are inserted at the synchronization points mentioned above, it is very likely that these
two errors are taking place in only several instructions away. Considering that soft
errors are rare events, this scenario is safe enough to ignore.

### 3.4 Error Correction

The error correction and recovery can be implemented in an error handler invoked on error
detection. The handler can implement any user-defined error correction technique. This code
is executed very rarely since soft errors are infrequent events. Therefore error correction can
be slow. The error correction has to be done in two different ways depending on where the
error checking code is inserted (as described above).

1. The error correction in the case when the checking code is inserted after each duplicated
   instruction can be accomplished by performing the operation a third time and utilizing
   majority voting. The error handler can use the call site program counter to identify
   and re-execute the instruction in error one more time and perform majority-based
   correction.

2. The error correction in the case when the checking code is only inserted at certain
   synchronization points is probably best accomplished by an intelligent program check-
   pointing mechanism.

### 3.5 Summary

This chapter proposed a compiler approach to detect soft errors through duplication using
SIMD features of modern processors. It was implemented in the LLVM compiler. The
implementation is flexible and portable.
The proposed technique can significantly benefit from architectural support. A single additional SIMD instruction that compares two elements of an SIMD register and generates an exception when they are not equal would reduce error checking code overhead to a single instruction. This will make it feasible to check each duplicated instruction and perform immediate error correction.
Chapter 4

Experimental Evaluation

This chapter describes the experimental evaluation of the proposed error detection technique and compares its performance, energy, and code size with prior instruction duplication techniques. LLVM was used (with the -O3 flag) to compile the original code and to add the proposed SIMD-based detection as well as the scalar duplication. Note that the -O3 compilation enables automatic vectorization. Only C/C++ is fully supported by LLVM, so only C/C++ benchmarks were used in the experiments. Note that prior work to which we compare (SWIFT) also used a subset of SPEC2000 benchmarks and subsets of two other benchmark suites.

4.1 Experimental Setup

Compiled benchmarks were executed on a system with an Intel Core i7-4770 processor and 8GB memory, running Linux 3.13.0 kernel. The processor supports AVX2 and all prior SIMD extensions. Hardware performance counters were used to measure the impact of the proposed approach on benchmark execution. The energy consumption results were collected via Likwid.
Figure 4.1: Performance slowdown of scalar and vector instruction duplication (no error checking added).

Figure 4.2: Energy overhead of scalar and vector instruction duplication (no error checking).
using the PKG counter for total energy consumed by the socket in application execution.

**Benchmarks.** The C/C++ benchmarks used in the experiments include floating point benchmarks from SPEC2000 and SPEC2006 suites and from seven smaller benchmarks: three benchmarks from SPECFP2000 (179.art and 183.equake, 188.ammp), five benchmarks from SPECFP2006 (433.milc, 444.namd, 447.dealII, 450.soplex, and 470.lbm), and IRSmk, Aobench, FFT, SpectralNorm, Blackscholes, Seidel-2D, and Linkpack. All SPEC benchmarks were run using their reference data sets.

Each benchmark was compiled in several ways for SIMD-based duplication: with no error checking (vector+dup), with vector error checking after each duplicated instruction (vector+dup+CE), and with error checking inserted only at synchronization points (vector+dup+CS). Three versions were compiled for scalar instruction duplication: with no error checking (scalar+dup), with error checking after each instruction (scalar+dup+CE), and with error checking at synchronization points only (scalar+dup+CS). All results are normalized to the original program without any duplication (base).

The compilation in “scalar+dup+CS” approach is similar to the approach implemented in SWIFT but without the signature-base error checking for control flow (recall that we also do not duplicate loads). Otherwise both approaches duplicated the same instructions. In addition, our vector instruction duplication also duplicated the library functions with vector intrinsic alternative in the LLVM front-end.

### 4.2 Instruction Duplication Only

This section presents comparison of performance and energy consumption for “pure” instruction duplication, i.e. duplication without error checking (e.g. comparison of results and branching to an error handler), for scalar and SIMD versions of benchmarks.
Performance impact. The performance slowdown for scalar mode and vector mode duplication relative to the baseline is shown in Figure 4.1. The average slowdown for the pure SIMD-based instruction duplication is 17% and for the scalar instruction duplication it is 85%. The vector+dup technique is 36.75% faster, on average, than scalar+dup technique. The SIMD duplication executes fewer instructions than the code with scalar duplication because duplication does not require additional instructions, except for a) additional vector broadcasts required to duplicate a value loaded into an SIMD register and b) any additional packing and unpacking of vectors required. When the baseline program was not vectorized using the widest registers (YMM registers on the processor used for experiments). This is because only half of the lanes of an SIMD register are used for computation, the other half is used for duplication. Therefore, one can expect a minor performance slowdown by replacing scalar instructions with their vector equivalents.

Specifically, a vector register element is “duplicated” by adding an SIMD instruction to broadcast the low word to the high word. For example, we add `insertelement` and `shufflevector` instructions after each load in the IR code. The back-end generates a `vpbroadcastq` instruction after the `vmovd` instruction in assembly whenever a value is loaded from memory. Second, the LLVM back-end does not generate vectorized library calls for some functions, such as `sin`, `cos`, etc. Therefore, LLVM back-end has to unpack the values in a SIMD register, then perform two scalar library calls sequentially, and finally, pack the two scalar results back into the SIMD register.

Also, AVX-2 does not support integer division and 32-bit integer to double precision floating point conversion. These instructions also require the unpacking and packing operations. These are the main reasons for performance slowdown of SIMD based duplication. The pure scalar duplication overhead consists of duplicate instructions and increased register pressure.

The impact of these additional SIMD instructions depends on a specific benchmark.
• Memory intensive benchmarks, such as IRSmk, FFT, Seidel-2D and 183.equake, require many broadcast instructions to duplicate the loaded value which partially degrades the performance. It explains why they almost show similar slowdowns to the scalar instruction duplication scheme where loads are not duplicated. There are also library calls \((\sin \text{ and } \cos)\) in the main function of FFT and 183.equake benchmarks. Combining these two reasons, the slowdown for vector+dup of FFT and 183.equake is 1.17x and 1.16x, respectively.

• Aobench not only contains unvectorized intrinsic library calls, \(\sin, \cos,\) and \(\text{rand}\), in one of the hot loops in the \(\text{ambient} \_\text{occlusion}\) function but also has many integer to double precision floating point conversions, resulting in a 1.26x performance slowdown.

• The \(\text{mm}_\_\text{fv} \_\text{update} \_\text{nonbon}\) function in 188.ammp accounts for around 80\% of the total execution time. It uses many broadcast instructions and integer to double precision floating point conversions, leading to a 1.27x performance slowdown.

• Many integer division and type conversion instructions in 444.namd contribute to 1.26x performance slowdown.

• IRSmk, SpectralNorm, Blackscholes, 433.milc, and 470.lbm have lower performance slowdown because they don’t require many additional vectorization instructions.

The impact of additional scalar instructions is different. Blackscholes slows down by 3.5x for scalar duplication for two reasons. First, roughly 2x instructions are executed compared to the baseline. Second, the register pressure increases dramatically in this case and leads to many more register spills. However, the performance overhead for IRSmk and Seidel-2d due to scalar instruction duplication is low because out-of-order instruction issue and ILP features of the processor overlap the execution of original instructions and their duplicates perfectly.
SIMD duplication in Linpack has a slowdown very similar to its scalar duplication counterpart for a different reason. The compiler automatically vectorizes the application very well (see Figure 1.1). We need half of the SIMD register lanes for duplication, so only SSE vectorization is performed prior to our SIMD-based instruction duplication pass. Thus, the performance slowdown of vector+dup is 1.66x. However, this slowdown is still less than scalar+dup as the compiler cannot perfectly vectorize Linpack, leaving some vector units for instruction duplication.

**Energy consumption.** The energy consumption of scalar and vector mode instruction duplication is shown in Figure 4.2, normalized to energy consumed by the baseline. On average, pure scalar instruction duplication and vector instruction duplication consume up to 1.91x and 1.11x more energy, respectively. The energy consumption caused by vector mode instruction duplication is quite small.
Figure 4.4: Energy overhead of instruction duplication with selective and full error checking.

### 4.3 Instruction Duplication with Error Checking

This section evaluates instructions duplication with error checking code added in two different ways: after each instruction (CE) or only at synchronization points (CS), such as stores and function calls. Figures 4.3 and 4.4 show normalized execution time and energy consumption for the differently compiled versions of benchmarks.

The scalar duplication plus checking runs, on average, 2.11x slower and consumes 2.14x more energy compared to the baseline when error checking is performed at synchronization points only (scalar+dup+CS). The average performance and energy overheads increase sharply when error checking is performed after each duplicated instruction (scalar+dup+CE), to 4.64x and 4.97x, respectively.

In comparison, the SIMD based duplication with error checking at synchronization points (vector+dup+CS) runs 1.32x slower and consumes 1.25x more energy, on average, over
Table 4.1: Standard deviation of performance and energy overheads.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Performance</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar+dup+CS</td>
<td>0.74</td>
<td>0.76</td>
</tr>
<tr>
<td>vector+dup+CS</td>
<td>0.30</td>
<td>0.22</td>
</tr>
<tr>
<td>scalar+dup+CE</td>
<td>2.48</td>
<td>2.57</td>
</tr>
<tr>
<td>vector+dup+CE</td>
<td>2.33</td>
<td>1.83</td>
</tr>
</tbody>
</table>

the baseline. Compared to scalar+dup+CS, its performance and energy overheads are, on average, 37.25% and 41.6% lower, respectively, compared to scalar mode. The SIMD based solution with error checking after each instruction (vector+dup+CE) results in the average performance slowdown of 3.64x and consumes 3.38x more energy.

One more observable benefit of the proposed technique is that it delivers more stable performance and energy consumption across all benchmarks. Table 4.1 shows that the standard deviation of the SIMD based performance slowdown is 0.30 (for vector+dup+CS) and 2.33 (for vector+dup+CE). The corresponding standard deviations for the scalar error detection approach are 0.74 and 2.48, respectively. The standard deviations of energy consumption for scalar+dup+CS and scalar+dup+CE are 0.76 and 2.57, respectively vs 0.22 and 1.83 for vector+dup+CS and vector+dup+CE schemes, respectively.

There are at least two reasons behind the above phenomenon: a) while auto-vectorization does take place it leaves sufficient vector resources for redundancy if it doesn’t vectorize an application perfectly, and b) not all of the benchmarks are able to benefit from the ILP capabilities of the processor for scalar fault tolerant techniques.

4.4 Code Size Overhead

Figure 4.5 shows the binary size of the different compiled versions, normalized to the baseline with no instruction duplication. Our scalar duplication does not apply to the IR loads, stores, branches, function calls. SIMD based duplication replicates the same type of instructions,
Figure 4.5: Binary size of scalar and vector instruction duplication with selective and full error checking.

plus instructions in the library calls that support SIMD intrinsic instructions in LLVM. The scalar code scalar+dup+CS is 2.52 times larger than the baseline while the SIMD based code vector+dup+CS increases the code size by 1.96x. The binaries generated by scalar+dup+CE and vector+dup+CE (checking on every instruction) are, on average, 5.77x and 4.731x larger, respectively.

4.5 Overhead with Memory Protection

We have studied the cost of our vector-based instruction duplication vs scalar duplication in the previous sections without protection of memory instructions, e.g. loads and stores. Memory protection has been largely omitted by most of previous works with the assumption that they are guarded by EEC. However, this might not be the case for many commodity processors and embedded processors. We proposed an approach to protect loads and stores...
using gather and scatter primitives, respectively. Recall that gather/scatter primitive loads/stores data from/to two memory locations that are away by an offset into/from two lanes of a vector register. Note that LLVM only supports masked gather and scatter for AVX-512 extensions on Intel. Now let us investigate the overhead in terms of performance and energy consumption caused by vector-based instruction duplication with memory access protection.

Intel started to support gather and scatter instructions from the Skylake architecture and LLVM started to support code generation for these instructions from 3.9 version. Here we used LLVM 4.0 to compile the benchmarks. These experiments were conducted on a Linux machine with an Intel Xeon Phi(TM) 7210 Knights Landing processor which supports AVX-512 SIMD extension. Although we experimented on Xeon Phi processor, the proposed compiler technique is generic and can be applied on any target processor with vector extensions, e.g. ARM, that support gather loads and scatters stores to enable vectorization of complex data structures with non-linear access patterns.

We compare the overhead of adding protection on memory loads and stores (e.g. scalar+mp) to that caused by the scalar instruction duplication. Note that the scalar duplication (scalar+dup) doesn’t actually protect memory accesses.

Figure 4.6 and Figure 4.7 present the performance slowdown and energy consumption for both the scalar duplication and by adding memory protection compared to baseline program (the original program). The overhead in terms of both performance and energy consumption increases significantly over the baseline benchmarks, e.g. 1.81x and 1.84x for the former and the latter, respectively. More instructions are required to protect memory loads and store as the gather and scatter primitives are adopted. These two primitives use 512-bit zmm registers as the operands. In order to effectively use these registers, proper mask registers (e.g. k0-k7) have to be set to indicate which lanes of a zmm register are actually effectively used. Usually, a few instructions are needed to set correct masking.

In addition, after performing gather operations, one has to move the useful data certain lanes
Figure 4.6: Performance slowdown of scalar and vector instruction duplication with load and store protection.

from a $zmm$ register to either a $xmm$ or a $ymm$ register for other operations that depend on the gathered values. Move the corresponding lanes from a $zmm$ register to fill the narrower register would also require several instructions. Figure 4.8 corroborates the above explanations. As we can see that the average dynamic instructions count for these benchmarks is now 2.2x more than the baseline program for vector+mp, while it is about 2.5x for the scalar duplication (scalar+dup). The dynamic instruction count boost is more prominent for memory intensive benchmarks, e.g. IRSmk, 183.equake, and 470.lbm, etc. For instance, the number of dynamic instructions goes up to 2.87x and 2.41x for 183.equake and 470.lbm, respectively. However, it’s still better than the scalar duplication where the performance slowdown and energy consumption are 2.02x and 2.08x over the baseline, respectively.
Figure 4.7: Energy overhead of scalar and vector instruction duplication with load and store protection.

Figure 4.8: Instruction count overhead of scalar and vector instruction duplication with load and store protection.
Chapter 5

Mitigation of Crypto Fault Attacks

Mitigations against fault-attacks on cryptographic primitives are well established in the embedded system domain, specifically when cryptographic implementations execute on simple microprocessors, such as smart cards. These are software only mitigations, they are highly effective when implemented correctly, and are portable, and, more importantly are appealing because they do not required changes in the hardware architecture underneath [114]. One of such mitigations, instruction duplication [9], is implemented at the instruction set architecture (ISA) level. Its effectiveness against both fault- and side-channel attacks for block ciphers and public key primitives has been shown and the mitigation is widely accepted in the community. Instruction duplication and triplication countermeasures (herein after "instruction redundancy") [9] are well established Instruction Set Architecture (ISA) mitigations against fault attacks. Instruction duplication executes each instruction of a cryptographic algorithm twice and compares the results from both runs to detect the occurrence of a fault. This countermeasure takes advantage of time-based redundancy. It assumes that the original instruction and its duplicate will not be affected by the same faults. Hence, the presence of a fault can be detected.
Albeit relatively more appealing in performance to cost ratio than more recent hardware countermeasures [78], instruction duplication suffers from more than $2x$ instructions (e.g., duplicates and error checks) when instructions are duplicated in a naïve manner. Thus the protected algorithms still suffer from high performance and (also) code size overheads. In addition, the extra instructions may also induce register pressure. Combined with these effects, as reported in [9], instruction duplication can cause up to $3.4x$ times performance slowdown.

Yuce et al. [114] have shown that instruction duplication, triplication etc, can be thwarted by leveraging artifacts of pipelined execution. Specifically, due to the asymmetry in the critical path of instructions, Yuce et. al. have shown the possibility of injecting a single or multiple glitches at certain pipeline stages, and that the fault propagates through the critical path of the affected instructions in a way to bypass instruction duplication (and triplication) countermeasure. To fully utilize the benefits provided by some form of instruction duplication, a more sophisticated mechanism is therefore necessitated.

This chapter proposes to enable operation duplication by leveraging at its essence the single instruction multiple data (herein after "SIMD") extensions of modern microprocessors. SIMD extensions are ubiquitous in most commercial general purpose microprocessors, and the major manufacturers, such as Intel, IBM, and ARM suppliers, are apt to build increasingly larger vector units in new processor generations. SIMD extensions use a distinct set of instructions and operate on wider registers to complete multiple operations in parallel. When it comes to cryptographic implementations, the customary use of SIMD resources is to increase performance by leveraging the maximum amount of data parallelism available in a cipher through mapping the algorithm statements into vector statements manually, e.g., in the OpenSSL library.¹ This is particularly true since cryptographic algorithms commonly contain vectorization-unfriendly flow-control-heavy kernels, which further inhibits the

work of even state-of-the-art compilers. For example, we used a customized Pin tool [59] to count the instruction mix in the kernels used by public-key cryptographic algorithms in the Libgcrypt library [3]. It turned out that no vector instruction was automatically generated for these kernels to perform operations like addition, subtraction, multiplication, etc. That is, although the scalar units might have high pressure during the execution of these algorithms, both the vector function units and vector registers are largely idle.

This chapter presents a compilation approach, CAMFAS, which harnesses these available SIMD extensions to mitigate fault attacks. The proposed approach attempts to gain instruction redundancy for free, achieving both operation and data duplication. In this regard, CAMFAS is new, because it differs from both the traditional approach to code vectorization (here in after “SIMDization”), as well as the traditional approach to introduce instruction level countermeasures against fault attacks. The process essentially migrates the execution of most instructions from the scalar unit to the vector unit, and effectively transforms instruction redundancy into data redundancy. For example, in the case of instruction duplication, instead of running the original instruction and its copy sequentially as it was adopted in the prior art, CAMFAS vectorizes these two instructions and packs them into a SIMD register for execution. Error checking is then performed on the vectorized instruction for fault detection. To the best of our knowledge, this is the first work that exploits SIMD extensions to protect cryptographic algorithms against fault attacks, and trade-off resistance against fault-attacks with throughput in the cryptographic operations.

CAMFAS is fully implemented and automated in the LLVM compiler infrastructure [48]. Similarly to the state-of-the-art time-based instruction duplication techniques, e.g., [9], our SIMDization based countermeasure doesn’t need to be aware of any detail about the algorithm implementation. In addition, it delivers the following improvements.

First, previous instruction duplication directly inserts countermeasures into the code obtained via disassembling the executable object code for redundancy purposes. In the cir-
cumstance that the cryptographic code is inaccessible, this ad hoc solution is perhaps the only way to duplicate instructions. However, most assembly cannot be duplicated trivially, e.g., when a register is present as both the source and the destination. An expert has to judiciously save the temporary result into an unused register so that the content in the reused register will not be overwritten by the redundant computation. Thus duplication at the assembly level needs a considerable amount of expert effort. Therefore, the work of Barenghi et al. [9] only covers a small fraction of assembly instructions in the AES block cipher.

Our SIMDization based instruction redundancy strategy resorts to compilation techniques to trade-off throughput (achievable by taking full advantage of the available vector length) with resistance to fault attacks. It focuses on the intermediate representation (IR) level, which is after the front-end optimizations, but before register allocation and code generation. Replicated instructions and error checking code are inserted automatically without any manual intervention during the traversal of an IR form. Then the fault tolerant code generation is left to the back-end of the compiler. Our approach avoids the tedious and error prone assembly programming, yet still preserving all compiler optimizations. Portability is another important benefit of introducing mitigations at the IR level, since the IR is normally target independent.

Second, CAMFAS converts instruction duplication to operation/data duplication so repeating executions are avoided for individual instructions. This conversion effectively exploits spatial redundancy at the granularity of an instruction rather than exploiting the temporal redundancy as in [9], which is vulnerable to the single fault injected at certain pipeline stages [114].

Third, even though additional instructions may be needed to pack/unpack data into/from vector registers, CAMFAS parallelizes the execution of the original instruction and its duplicate thus leading to the significantly reduced code size and better performance.
Fourth, CAMFAS has lower register pressure because no extra registers are needed by the replicate.

The same method has been successfully applied to a series of benchmarks in the previous two chapters to protect against soft errors. However, this chapter mainly focuses on cryptographic algorithms, we have also implemented protection for memory accesses, e.g. loads and stores, using gather and scatter primitives. In addition, the source of the errors for cryptographic algorithms and the general programs are different. For example, the faults in general programs are caused by transient errors in hardware that are propagated to affect the output of execution, and the faults in cryptographic algorithms are usually injected intentionally by malicious hackers. Therefore, they may only focus on some small portion or even certain program points of a cryptographic algorithm. To mimic this effect, we injected errors in both the register files and memory locations to evaluate the effectiveness of our approach.

In our experiments, we have applied CAMFAS to cryptographic algorithms in the Libgcrypt library. The fault injection experiments demonstrate that our approach is able to deliver nearly full fault coverage with much reduced overhead. Furthermore, on average the mitigated cryptographic algorithms execute 2.2x slower and impose an average of 26% more dynamic instructions compared to unmitigated code.

5.1 Fault Model

This chapter considers an attacker capable of injecting glitches, as in [113, 29], which could corrupt data and/or instruction (mainly instruction skipping). The fault model is defined as the time, location and type of the injected fault in a cryptosystem, or more in general in a piece of software. The time and location of the fault is defined by the assumptions on the

---

2https://www.gnupg.org/software/libgcrypt/index.html
attack strategy and the targeted cryptographic algorithm [15, 12, 42]. In the case of data corruption case, the fault types that apply to our approach include: For data corruption, the main focus of this work, fault types that apply to our approach include:

- **Single Random Bit-Flip**: The fault injection induces bit-flip in a random position in a register.

- **Multiple Random Bit-Flips in Different Registers**: Multiple faults can be injected and introduce independent bit-flips at a random location in each register.

- **Multiple Random Bit-Flips at different elements in the same SIMD Register**: Multiple faults are injected into the same SIMD register but at different elements of it - it is easy to prove that this type of fault can occur with a negligible probability which further decreases with the vector length. Multiple faults that are injected to identical positions different vector lanes are not considered in this work because it is much more difficult to inject these type of faults [9].

"Identical locations" are referred to the same bit positions in each element of a SIMD register. For example, the first and the 65th bits are identical locations in a 128-bit SIMD register when the manipulated data and duplicated and its type is 64-bit.

We purposely factor out the case of simultaneous faults that cause systematic bit-flips in the exact same position in two elements of a SIMD register, e.g., the first bit and the 65th bit of a 128-bit xmm register. On one hand, let $x\%$ be the probability of an occurrence of two random faults in a SIMD register. The probability that these two faults affecting the same position in two different elements of a vector in a $n$-bit SIMD register is $\frac{n/2}{2^n} \cdot x\% = \frac{x\%}{(n-1)}$. Such a probability is 1 to 2 orders of magnitude smaller than $x\%$ for vector registers since the vector length is at least 128 bits and it doubles in new extension generations. On the other hand, the ability of injecting faults with a spatial precision is outside the threat model.
And even if it is part of the thread model, the technology node of modern general purpose embedded and high performance systems poses difficulties to the process.

An attacker can skip an instruction or replace it with an effective \texttt{nop}. The error checking code, or particularly the error checking branch instruction, might be targeted by adversaries to thwart the inserted countermeasures. We refer to literature to mitigate this case. For example, in [24], instructions are rearranged so that the vulnerable branch instruction is followed by a \textit{default-fail} error handling module. In the case of instruction corruption, we also assume that the attacker can skip an instruction or replace it with an effective \texttt{nop}. The error checking code, or particularly the error checking branch instruction, might be targeted by adversaries to thwart the inserted countermeasures. However, this class of errors can be protected by the means proposed in [24], where instructions are rearranged so that the vulnerable branch instruction is followed by a \textit{default-fail} handling module. With this modification, the code will immediately enter the error handler once the branch instruction is bypassed. Other instruction level approaches that replace one vulnerable instruction with a sequence of fault-tolerant instructions to prevent from bypassing instructions are also orthogonal to our work [65, 76].

\section{Evaluation}

This section provides an experimental evaluation of \textsc{CamFas}. We will first inspect its capability in detecting injected faults and then study the overhead in terms of performance and the dynamic instruction count. LLVM 4.0 was used to compile the Libgcrypt library and to add mitigations against fault-attacks. The following cryptographic algorithms are investigated: RSA; DSA; ELG; and ECC, this last includes ECDSA, ED25519; and GOST. We duplicated the kernels that are heavily used by the cryptographic algorithms, i.e., the ones under the \texttt{mpi} directory in the Libgcrypt library. These kernels perform all the required arithmetic
operations for each of these four cryptographic algorithms, e.g., addition, subtraction, multiplication, division, modulo, etc. All the results are normalized to the originally unmodified program which is also referred to as the *baseline* in this section.

### 5.2.1 Experimental Setup

All versions of the compiled benchmarks were executed on the Intel x86_64 ISA with AVX-512 SIMD extension. Specifically, the processor model is Intel Xeon Phi(TM) 7210 Knights Landing processor and 16GB memory, running Linux 4.4.0 kernel. While we experimented on Xeon Phi processor, the proposed technique is generic and can be applied on any target processor with vector extensions. Nowadays fault attacks are also viable on high performance processors, as the technology has improved for glitch or laser injection, e.g., via VC glitcher.³

![Graph showing fault distribution](image)

**Figure 5.1:** The breakdown of faults injected on each type of instruction for different mechanisms: no fault detection (N), duplication without gather and scatter instructions (O), duplication with gather and scatter instructions (W).

³[https://www.riscure.com/security-tools/hardware/vc-glitcher](https://www.riscure.com/security-tools/hardware/vc-glitcher)
5.2.2 Fault Coverage

Now we evaluate the effectiveness of our SIMD-based fault detection technique. The fault coverage of the original and the vectorized binaries (both with and without gather/scatter instructions) are analyzed by injecting faults to the corresponding binaries. PIN [84] is used to achieve this in two steps. First, we collect the full program trace via a customized Pin tool. This step records all the executed instructions associated with their instruction pointers (IP) and read/write registers for every iteration of the specified functions (kernels) in the hardened part of the program. Second, an instruction at a random iteration is picked from the trace file before the program starts execution again. An operand register is then arbitrarily selected for fault injection, and finally a single bit in a random byte of the register is flipped during the execution of the program. Although only single-fault model is simulated in experiments, our work is able to detect multiple flipped bits because they will produce the same effect as a single random fault. Recall, it is very difficult to inject identical faults to both the original and its redundant copy.

We modified register files, including general purpose registers and floating point registers, while leaving the memory contents untouched. We also reproduced the scenario when one bit of a memory read address is flipped, to emulate errors in the memory address for load instructions. Similar fault injection has been adopted in [88].

Our technique is not limited to protecting arithmetic units and registers. It is capable of detecting faults in other microarchitectural units as well if the errors will finally propagate to affect the values in registers. The faulty program continues until completion and the outcome of each run is recorded. 1000 faults were injected in each cryptographic algorithm execution to collect the statistics. The results are classified into one of the following four categories by comparing to the known good outputs (e.g., the result produced by the unmodified original binary).
- **Detected**: The program terminates due to the fault being detected by our vector based error detection technique.

- **Incomplete**: The injected fault causes abnormal behavior of the target binary, i.e., it may result in the infinite execution as the error makes the loop termination condition not hold ever, or a program with inserted countermeasures fails because of segfaults (e.g., the injected error causes invalid memory access) or other faults, such as double free or corruption (i.e., the injected error propagates to cause double free of a portion of the memory), floating point exception (e.g., divide by zero), and bus error (i.e., the injected error leads to misaligned address access).

- **Masked**: The faulty program completes normally and produces correct encrypted/decrypted results. In this case, the injected error didn’t corrupt the program due to application level or architectural level masking. As we will present later, some injected errors are masked in Libgcrypt mainly due to the wide use of bitwise operations in the algorithms.

- **Corrupted**: The faulty program finishes execution but it produces a different encrypted/decrypted text compared to the original value. In this case, the program was not aborted during execution but verification signaled a failure in the end. This is the case where an attack is successful since the attacker has access to the faulty results.

Libgcrypt library provides a series of self tests to make sure that the value computed at each stage is valid for the cryptographic algorithms. It signals a failure when a mismatch is found between the modified data and the original data. This case is also classified into the **corrupted** category because neither our mechanism nor the system detects it. To guarantee that each run could be finished in a reasonable amount of time, We only used 1024-bit keys for RSA, DSA and ELG, 192-bit and 256-bit inputs for ECDSA, and 256-bit input for GOST. A run is treated as **timeout** and thus goes to the **Incomplete** category when it
Figure 5.2: Fault coverage by each cryptographic algorithm among different mechanisms: no fault detection (N), duplication without gather and scatter instructions (O), duplication with gather and scatter instructions (W).

Cryptographic algorithms

- RSA
- DSA
- ELG
- ECDSA
- Ed25519
- GOST
- Mean

Corrupted, Masked, Incomplete, Detected
executes at least 5 times longer than producing the masked results.

Let’s first look at the distribution of faults that have been injected into different instruction categories. Figure 5.1 shows the breakdown of faults injected in each type of the instructions for three configurations, baseline with no fault detection (N), vector based duplication with gathers/scatters (W), and vector based duplication without gathers/scatters (O). Four major instruction categories are investigated, 1) arithmetic instructions including logic and rotation operations, 2) memory loads (stores are excluded since checks are inserted before them to guard the validity of address computation), 3) comparison operations including those inserted for fault detection, and 4) the "Other" category including instructions that perform shuffle operations before each comparison for error detection as described in Chapter 3.

Note that the last category doesn’t apply to the baseline, since no such operation exists in the original program.

Figure 5.1 illustrates that the baseline with no fault detection (N) has 0 percent of instructions sitting in the “Other” category for all of the cryptographic algorithms. For the baseline, an average of 70% and 20% faults are injected to the arithmetic instructions and the load instructions, respectively. Only 10% of faults are injected to comparison instructions. However, for CAMFAS with gathers and scatters, about 60% and 8.4% faults are injected to arithmetic instructions and load instructions on average, respectively. Compared to the baseline, these two numbers are decreasing because other instructions are introduced as countermeasures. The percentage of faults injected to comparisons increases by 12% for CAMFAS with protection of memory computation. This number is very close to the percentage of errors injected to “Other” instructions (e.g. 9.2%) because every error checking code requires one shuffle instruction. We can also observe that the percentage of errors injected to each instruction category for our duplication without using gather and scatter instructions is close to the case of using gather and scatters. This is because the number of instructions for these two mechanisms are close to each other despite that they employ different sets of instructions to
fetch data from memory, i.e., the former uses broadcast and the later uses gather combined with a few other instructions to prepare the mask.

Figure 5.2 associated with Table 5.1 shows the effectiveness of our vector based error detection approach in detecting injected faults. As it is tabulated in Table 5.1, CAMFAS with memory address protection yields negligible corrupted results. These incorrect results are mostly caused by the errors injected to the error checking code since this portion of code is not protected. The percent of corrupted runs goes up to 5.37% when we don’t check memory addresses. It is significantly reduced compared to the baseline where an average of 23.35% runs are corrupted. In the worst cases, the corrupted results could be up to 38% (ED25519).

<table>
<thead>
<tr>
<th>Approach</th>
<th>Detected</th>
<th>Incomplete</th>
<th>Masked</th>
<th>Corrupted</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0</td>
<td>61.65%</td>
<td>15%</td>
<td>23.35%</td>
</tr>
<tr>
<td>O</td>
<td>45.37%</td>
<td>44.43%</td>
<td>5.83%</td>
<td>5.37%</td>
</tr>
<tr>
<td>W</td>
<td>46.5%</td>
<td>49.72%</td>
<td>3.42%</td>
<td>0.36%</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of fault coverage rates of each category among no fault detection (N), duplication without gather and scatter (O), and duplication with gather and scatter (W).

Not only is our technique effective in reducing the number of corrupted results, but also it can significantly lower the incomplete results from 61.65% to less than 50%. This effect is particularly marked in RSA where the rate of incomplete execution drops from 77% in the baseline to 49% in the binary with countermeasures using gather and scatter instructions. This is because error checking code is added before stores and branches to validate the correctness of the store destinations and branch conditions. This error checking may have taken effect before abnormal program behaviors actually took place. While effective in protecting the binary from being corrupted, duplicating memory access addresses also moderately raises the number of incomplete executions. The reason is the single-bit fault in memory address would usually result in the access of unmapped memory region which leads to a segfault.

Sometimes a binary could still finish execution despite the injection of an error at a certain run. This can happen in cryptographic algorithms since some of the developed kernels contain operations such as and, or, etc. The result of these operations might not depend on
Cryptographic algorithms

Figure 5.3: The slowdowns of error detection enabled cryptographic algorithms to compute private and public keys compared to the original ones without error detection for different input sizes.

the value of the operand where a fault is injected. However, the number of masked runs falls from 15% in the baseline to 3.42% and 5.83% in our duplicated versions with and without gathers/scatter, respectively.

Most of the decreased numbers from the above three categories come into the detected category, i.e., more than 45 percent of the faults are detected by CAMFAS. The difference between the error detection rate of using and not using gather/scatter instructions is minimal, e.g., less than 1.2%. This is because a majority of errors injected in memory addresses are turned into segfaults as discussed above.
5.2.3 Fault Attack Prevention Discussion

There are different categories of fault attacks on cryptosystems that are divided based on the assumption on the fault model. CAMFAS detects and prevents cryptosystems from the following categories of fault attacks:

- **Differential Fault Analysis (DFA):** Fault Attacks in this type consider a hypothesis on the fault injection. This hypothesis is usually a single bit-flip, random byte, or random bit-flip. The adversary is then required to capture correct and faulty cipher texts and reverse engineer the differential to obtain the secret key. CAMFAS detects DFA attacks because based on the results provided in Figure 5.2, the number of “Incorrect” outputs has reduced from 23.35% to 0.36%.

- **Fault Sensitivity Analysis (FSA) [54]:** Unlike DFA, for FSA to obtain the secret key, there is no need to have the value of the faulty cipher text. FSA relies on the correlation between the values of an intermediate variable and the Fault Sensitivity point which is the time that fault occurs for the first time. CAMFAS has a high detection rate for this type of fault attack since it terminates the program in case of detection. In case of *Incomplete*, the adversary can capture this point and correlate it to the intermediate value, since incompleteness is a sign of Fault Sensitivity Point. However, we believe that since capturing an *Incomplete* condition cannot be executed accurately. So in case of the FSA attack, the adversary is required to inject faults into more traces.

- **Differential Fault Intensity Analysis (DFIA) [29]:** Unlike other fault attacks, DFIA does not rely on a fault model hypothesis. It relies on the bias of fault behavior and the differential of faulty outputs. Since CAMFAS prevents the faulty output from being propagated, it is able to effectively prevent DFIA.

- **Single-Glitch Attacks:** [114] has proposed some hardware based microprocessor fault attacks that rely on different scenarios of fault injection. Yuce et.al, have proposed
a framework, by inspecting fault sensitivity characterization, in which the adversary is able to find scenarios to thwart redundancy techniques. In case of these attacks, as provided in Section 5.1, the possibility of corrupting the original and its duplicate is low which eliminates the case of scenario 1. We still believe that Scenario 2 and 3 might be possible based on the underlying microprocessor platform. However, since the comparison of the original and the duplicate copy is at the IR level, an accurate fault injecting into the vulnerable points in the pipeline to modify comparison instruction to **nop** will be more difficult.

### 5.2.4 Fault Detection Overhead

The performance and code size overhead caused by the virtually full coverage of our inserted countermeasures will be examined in this section.

![Figure 5.4: The dynamic instruction count overhead for duplication with and without gather/scatter instructions.](image)

Cryptographic algorithms

Figure 5.4: The dynamic instruction count overhead for duplication with and without gather/scatter instructions.
Performance overhead. Figure 5.3 shows the performance slowdown of our vector based error detection on RSA, DSA, ELG, and ECC algorithms with different input sizes compared to the baseline. The input sizes for RSA, DSA, and ELG are 1024, 2048, and 3072 bits, respectively. The input for ECDSA are 192, 256, and 384 bits. GOST is tested with 256- and 512-bit keys. Two groups of experiments were conducted. One duplicates load and store addresses using gather and scatter alternatives. The other, instead of using gathers and scatters, loads the content in an address and broadcasts it to fill a SIMD register. In Figure 5.3, a “private/public with g/s” bar represents the slowdown of a vectorized benchmark with gather/scatter instructions over the baseline counterpart when computing the private/public key. A “private/public without g/s” bar indicates the slowdown without using gather/scatter instructions.

The average slowdowns of the instruction duplicated cryptographic algorithms are 2.2x for computing both the private and the public key when gather/scatter instructions are used to duplicate the memory addresses. The slowdowns are mainly attributed to the following facts.

First, there is no vector form of integer division and quadword multiplication instructions. Therefore, all these division and multiplication operations are performed as scalars that require unpacking the upper and bottom quadwords from a SIMD register. After unpacking, two scalar \texttt{divq/mulxq} instructions are executed sequentially to obtain the upper and bottom quadword values. These two values are packed into a SIMD register using \texttt{vpunpcklqdq} instruction in the end. The whole process needs at least 7 instructions, therefore causing high performance overhead.

Second, extra instructions are needed for error detection beside comparison and test instructions. For example, we have to swap the upper and bottom quadwords before a comparison for error checking since there is no direct way to check the equivalence of them.
Third, most vector instructions are more expensive than their scalar counterparts on Knights Landing processors in terms of latency, although the number of micro operations from the decoder might be the same. For instance, all vector instructions have a latency of at least 2 clock cycles on the Knights Landing processor, while their scalar alternatives on general purpose registers usually have a latency of 1 clock cycle [28].

The overhead reduces to an average of 1.7x for both private and public key computation if load/store addresses are not duplicated (except the ones computed by previous arithmetic operations) using gather/scatter instructions. This is because gather and scatter instructions are very costly on most of Intel processors, i.e., \texttt{vpgatherqq} instruction needs 18 and 15 clock cycles on Skylake and Broadwell [2], respectively. At least the same latency would be expected on the Knights Landing processor because it needs 6 micro operations (\texttt{\muops}) vs 5 on Skylake [28].

**Dynamic instruction count overhead.** Figure 5.4 exhibits the total instruction count for our instruction duplication techniques with and without gather and scatter instructions, where the numbers are normalized to the baseline. The normalized instruction counts show a geometric mean of 1.26 when gather/scatter instructions are used to duplicate memory addresses, while this number is only 1.13 when gathers/scatters are not used. However, traditional instruction duplication would generally require at least twice as much as instructions for duplication. Therefore, our vector based instruction duplication approach is able to efficiently detect attacks only at an insignificant instruction count overhead across the public key cryptographic algorithms. Furthermore, these numbers provide more justification for some facts that are given in the previous performance overhead experiment.

First, using gathers and scatters to duplicate memory address computations only requires 13% percent more extra instructions (e.g., \texttt{kmovw} to set write mask registers) compared to the one that doesn’t protect memory address computations. Recall that, compared to the baseline, the average performance slowdown values of the versions where memory address
calculations are hardened and not hardened are around 2.2x and 1.7x, respectively. The results altogether corroborate that gathers and scatters are expensive operations that primarily contribute to the additional performance degradation.

Second, the dynamic instruction count grows disproportionally to the increase in performance overhead shown in Figure 5.3. This is consistent to the fact that integer vector instructions are generally much more expensive than their scalar counterparts on the Knights Landing processor.

Another interesting observation is that the variations of the dynamic instruction count for each bar in Figure 5.4 are small for both cases. For instance, the instruction count overhead ranges from 21% to 29% percent from duplication with gathers and scatters, and it is from 6% to 22% when gathers/scatters are not used.
Chapter 6

Related Work

Error detection mechanisms can be classified into different categories in terms of the ways they used for protection. Recent papers, developed independently of our own, proposed either software-based mechanism, e.g., EDDI [73], SWIFT [87], and Shoestring [26] with instruction duplication, hardware-based techniques such as RMT [67], SRTR [107], CRT [32], and [51], and hybrid schemes for error detection. The common of these proposals is to exploit either space or timing redundancy to detect errors with various tradeoff options.

This thesis focuses on using software instruction duplication with vectorization to reduce the soft error detection overheads in terms of performance, energy, and code sizes. The whole framework is implemented in the LLVM compiler without additional hardware modification. Previous work proposed hardware, software, and hybrid error detection techniques to address the soft error issue.
6.1 Hardware level

Two forms of redundancy are often exploited for error detection at this level, *structural redundancy* and *temporal redundancy*. DMR (dual-modular redundancy) and TMR (triple-modular redundancy) are structural redundancy based mechanisms to achieve virtually 100% error coverage by executing each operation on two or three exactly the same hardware units. However, these approaches are often deployed for mission-critical applications at a considerably high hardware cost. For example, targeting seven-nines (99.99999%) reliability, HP Integrity NS16000 and NS14000 servers provided DMR and TMR to tackle single hardware failures [11]. IBM S/390 G5 fully duplicated its execution units [98] and SIEMENS SIMATIC S7-400H offered redundant CPUs to protect them against failures [4]. DIVA [6] is a heterogeneous DMR alternative that uses a simplified core to monitor the operations performed by the more complex core. DIVA works well on large speculative RISC processors but less efficient on processors with little speculation. At finer granularity level, such as circuits, hardened circuits are designed for resiliency in high-radiation environments with the penalty of longer clock-cycle and about twice chip areas [82, 64].

AR-SMT [89], Simultaneous Redundant Multi-Threading (SRMT) [85], SRTR [107], CRT [67], CRTR [32], and [93, 75, 63] are typical papers proposed to use temporal redundancy for transient error detection. These papers mainly focused on using redundant multithreading to detect soft errors. Two threads are always spawned to run an application. A leading thread is used to run the actual program and a trailing thread is used to check the correctness of the values. AR-SMT [89] expanded the idea of RMT on SMT processors. SRMT [85] improved the performance of AR-SMT where checkers were performed before stores. CRT [67] and CRTR [32] worked at the chip-level, but CRTR implemented recovery for CRT. These approaches need extra hardware support to validate the comparisons between the leading thread and the trailing thread. RMT-based approach fits well for control-intensive where idle processor execution resources can be utilized for redundancy but are not directly
applicable for compute-intensive processors where resources are rarely idle.

[91, 83, 80] are also temporal redundancy techniques by exploiting the idle execution resources to detection errors for control-intensive application. However, as execution resources are rarely spare for compute-intensive, applications, these techniques are hence not directly applicable.

Myriad architectural mechanisms are also proposed to detect errors in different hardware components. At the architectural level, a simple checker, like DIVA [6], module can be designed to detect errors during the execution. This generally induces reasonably low performance and hardware cost. This scheme is quite viable for complex processor with deep pipelines and control-intensive superscalar, but it works much less effectively when applying to compute intensive processors as the computational engine itself is as simple as the checker. In this scenario, full hardware duplication of a major portion of the processor might be even more applicable as it was designed by the lockstepped IBM G5 processor [97].

Target reducing the high hardware cost, researchers have proposed schemes to only introduce some small augmentations to the processor pipeline [25, 13]. RAZOR [25] is a method proposed to detect and correct timing error dynamic. It’s goal is to save power by tuning the processor supply voltage. While it achieves very high error coverage, duplication of the large portion of the processor results in prohibitively high hardware overhead as the design is irrespective of the presence or absence of faults. For example, researchers have shown that protecting the register file with ECC is extremely costly in terms of both power [79] and performance [104].
6.2 Software level

While the software-based technique combined with vectorization for error detection is new, vectorization itself has been gathering a plenty of research interests in the past few decades. Modern microprocessors feature SIMD vector as crucial units, e.g., X86 SSE/AVX, ARM NEON, and PowerPC AltiVec, making SIMD vectorization popular in the compiler community [41, 70, 36]. Most of the techniques for vectorization are based on analysis and automatic transformation of loops, control/data flows, ILP vectorization through loop unrolling, superword-level parallelism (SLP), or speculation, etc. However, no previous work attempted to pack the redundant instructions in vector units for error detection. These vectorization techniques are orthogonal to our error detection/correction. For example, we could rewrite a program with SSE and AVX intrinsics to vectorize its hot loops so that instruction duplication/triplication is implemented without substantial performance and energy overhead. We could also further improve the parallelism of the vectorized applications by using these existing vectorization schemes.

Software based error detection approaches, e.g., [37, 38, 14, 50, 27, 40], etc, are attractive because they are portable to most systems without modifying the underlying hardware. For example, our proposed technique is applicable to modern processors integrated with SIMD units. However, software approaches often come with performance and energy overhead as replication and re-execution are required. In addition, software-only techniques generally cannot provide the same level of reliability as hardware approaches since they are unable to examine microarchitectural state [86]. Shirvani et al. [94] proposed to design a software-only technique to enable ECC for protection of memory data.

Compilers are commonly employed to automate error detection by intelligently inserting duplicates and checkers [44, 73, 87, 19, 47]. Khudia et al. [44] proposed error detection for soft computing applications using different protection schemes, e.g. traditional scalar
instruction duplication, value checks, and no protection, for different computations. Carbin et al. [18] proposed a programming language, Rely, that enables developers to reason about the quantitative reliability of an application. Wadden et al. [108] proposed an software approach to implement RMT on GPU hardware. IPAS [47] is an instruction duplication technique that protects scientific applications from silent data corruption (SDC) in their output. Their approach introduces partial redundancy to protect only code that are error prone. Partial duplication is efficient in reducing performance overhead, but it scarifies error coverage.

A few work relied on using symptoms such as fatal traps, cache miss, TLB misses [110], branch misprediction, value locality [55], behavior of program invariants [81, 56], and application aborts to detect errors [26, 45]. Shoestring [26] performed compiler analysis to identify and replicate only more vulnerable instructions. [45] improved Shoestring by using profiling information and they took into account anomalous microarchitectural behaviors for error detection. Symptom-based error detection techniques impose very little performance overhead compared to the full duplication techniques, but it scarifies error coverage due to only partial duplication. As the number of symptoms increases, the performance overhead also increases but the fault coverage starts saturating [44]. Racunas et al. [81] explored Perturbation Based Fault Screening (PBFS) by observing program invariants to achieve better coverage than the aforementioned symptom based error detection methods. However, true value changes in the PBFS approaches may cause false positives which unnecessarily trigger pipeline rollbacks and render high performance and energy overheads; and true faults may go undetected resulting in loss of coverage [69]. Compared to symptom-based techniques, we consider full instruction duplication for almost full error coverage.

Some automated frameworks have been proposed by using compilers to insert fault detection code (e.g. replicate instructions, comparisons, and checkers, etc) [71, 68, 57, 112]. These methods are generally not intrusive to the hardware design as the instructions are duplicated
by compilers automatically and the implementation and hardware details are transparent to software developers.

Among all these solutions, EDDI and SWIFT are most closely-related to our proposed solution in the context of full instruction duplication. Our technique differs from them in the following aspects.

- Both EDDI and SWIFT were developed for ILP-friendly processors, e.g., SWIFT was running on Intel Itanium processor, to take advantage of ILP provided by interleaved instructions. Our solution targets most modern processors by exploiting the idle vector resources for redundancy.

- The SoR (Sphere of Replication) of EDDI contains the whole processor and the memory system. SWIFT’s SoR excludes the memory subsystem, but our approach protects loads and stores using gather and scatter primitives, respectively. In addition, this work duplicates some library functions if they have SIMD intrinsic alternatives in LLVM.

- Our solution incurs much lower performance and energy overheads. It also reduces the code size significantly.

- Both EDDI and SWIFT assumed that memory is error-free and therefore they are our of the SoR. However, we approach doesn’t have this assumption. It duplicates loads and stores using gather and scatter primitives. Therefore, our approach achieves higher coverage for protecting processors.

- Last but not least, our approach is implemented at the IR level, making it more portable to different processors.

In addition, EDDI and SWIFT more target ILP-friendly architecture and [26] considers the protection of only global stores to trade off performance and error coverage. For EDDI,
comparisons are only inserted right before storing a register value back to memory or determining the branch direction. If an error is happened in the intermediate computations, they are not able to detect it until the error propagates down to these program points. By then the register that keeps the original value might have been allocated and released many times (assuming renaming), so recovery is not possible except with check-pointing. However, our approach can initialize error recovery immediately by repeating the computation one more time if an error is detected.

[46] also exploited instruction level redundancy using the SIMD feature of modern processors. The scheme was evaluated with a number of multi-threading applications and showed consistent results to [20].

**Control Flow protection** Signature based techniques are the most broadly adopted software-only techniques for control flow protection where each basic block is embedded with certain signatures/assertions at compile-time. Let’s briefly describe the basic idea of the signature-based control flow protection scheme, CFCSS, that was originally proposed in [72], as shown in Figure 6.1. A unique signature $s_i$ is assigned to each basic block in a program during compilation. Meanwhile, a shadow general purpose register ($G$), initialized to the signature of the very first basic block (e.g. $s_1$) when a program starts execution, is allocated to track the signature of the block that is currently executing and detect the errors.

![Figure 6.1: Signature based control flow protection. $G = s_2$ at the dest would hold if the control flow transfers correctly.](image-url)

$G = s_1$

$G = G \oplus d_2$

$br G \neq s_2$ error

$G = G \oplus d_2$

$br G \neq s_2$ error

$s_2$

$s_2 = s_2 \oplus s_1$
The value of $G$ is calculated and updated whenever the program branches from one basic block (e.g. $src$) to another basic block (e.g. $dest$) by computing the $xor$ of the $G$ and the result of the $xor$ between the signatures of the $src$ basic block and the $dest$ basic block. The output of the $xor$ operation should be equal to the statically assigned signature of the currently executing basic block ($dest$) if the branch was correctly directed. A comparison will be performed at the current basic block to verify the correctness of the control flow. The fault is detected once the comparison yields a mismatch. That being said, the flow was incorrectly taken place. Figure 6.1 illustrates the case where only one source basic block exists. For the more complicated scenario where multiple nodes share multiple branch-fan-in blocks, [71] proposed to adjust signatures at run-time to avoid aliasing.

Signature-based techniques have to introduce extra instructions for each basic block so that the current signature could be computed and updated. In addition, a few more instructions are necessary for comparison. These instructions will cause code size and performance penalty. Several other methods have been proposed to reduce the performance overhead [43, 16].

### 6.3 Hybrid approaches

While most of the aforementioned schemes focus on either hardware or software, some papers proposed to use hybrid approaches [86, 58, 62]. CRAFT [86] provided reliability using a software-hardware hybrid approach where duplicated instructions and error checking codes were inserted with software methods and additional hardware (similar to RMT) was used to achieve higher error coverage. CRAFT provided better reliability with less performance loss than most software-only techniques but requiring hardware modification. Argus [62] is a hybrid approach to protect four types of invariants, namely control flow, data flow, computation, and memory. Hardware units were used to generate invariants online and perform
comparisons between the invariants generated at runtime against the information collected from the compiler to detect faults. mSWAT [90] is also a symptom-based solution with little hardware support to detect anomalous software behaviors. Relax [22] is an architecture that exposes timing errors to software systems rather than handling them in hardware. It provides semantics similar to try/catch to offer reliability though a software/hardware hybrid approach. The goal of Relax is to obtain fault tolerance with relative low power through taking advantage of some code that is resilient to errors.

6.4 Fault Attacks and Mitigation

Recent prior art focused on specific hardware approaches to implement fault detection and mitigation [8, 113]. Hardware based techniques usually have to duplicate the hardware circuits, repeatedly execute a computation, and verify the results from both computations using a specific hardware unit. This type of countermeasures is quite costly in both application performance and hardware areas, plus end users are generally not able to modify off-the-shelf hardware.

In contrast, software countermeasures provide more flexibility and portability than hardware techniques as they don’t require any underlying hardware modification, hence increasingly receiving more research attention. Prior art on software based countermeasures propose mitigations either at the algorithm level [61, 42] or at the instruction level [9, 78] to hinder consistent fault injection.

Algorithm level countermeasures run a cryptographic algorithm twice and then compare the outcomes of both runs to check their equivalence. A fault is detected once a mismatch is signaled. These fault detection mechanisms focus on a coarse granularity, e.g., high level algorithms. The mitigations are not aware of the low-level architectural and hardware details.
While easy to implement, countermeasures at this level are prone to break by injecting identical faults to the same data across repeated executions [114].

With the assumption that injecting consecutive faults in subsequent instructions in a single cycle is infeasible, instruction level fault detection techniques concentrate on a much finer granularity, e.g., each individual instruction. These mechanisms attempt to counteract faults through duplicating instructions, repeating execution, and comparing the results from the original instruction and the redundant one. Instruction duplication is believed to be able to reach full error coverage, plus it is generally quite portable. However, the downside of it is the relatively high performance overhead, e.g., more than 3.4x in [9], and instruction size overhead since they need to execute at least as twice as many instructions. Another side-effect of instruction duplication is the increased number of registers which may bring high register pressure.

Considering the pros and cons of instruction duplication, we take a step forward to make these techniques less costly but still preserve their benefits by taking advantage of SIMD features from modern processors. Rather than duplicating and executing two identical instructions sequentially, our approach vectorizes the original instruction and its replicate using a SIMD register. It effectively converts operation duplication into data duplication, therefore obtaining fault tolerance with minimal overhead.

Pabbuleti et al. has looked into SIMD units to accelerate modular multiplications in prime fields [74], but they didn’t address the fault tolerance problem. Chen et at. proposed a compilation framework to utilize SIMD resources for fault tolerance [21]. They attempted to protect processors against soft errors by duplicating instructions into SIMD registers, but no memory address computation was protected. To our best knowledge, this is the first work that trades off performance and fault countermeasures using SIMD units.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

Manufacturers scale down the feature size of transistors to provide faster, cheaper, and more energy efficient chips. However, this trend also poses significant reliability challenges in modern microprocessors as it makes the transistors more susceptible to soft errors.

This dissertation proposed a compiler approach to detect soft errors through instruction duplication using SIMD features of modern processors. Initially, we manually vectorized kernels of a number of small benchmarks and verified that SIMD-based error detection causes much lower overheads compared to its scalar counterpart. Then we implemented SIMD-based error detection technique in the LLVM compiler as a standard-alone pass that works at the intermediate level. The implementation is flexible and portable, and does not require any hardware modification. This saves considerable chip area and has no impact on cycle time.

The performance, energy, and code size of compiled benchmarks were compared to previ-
ously proposed scalar duplication. The results show that, if error checkers were inserted at synchronization points only, the SIMD based soft error detection leads to a significantly lower overhead compared to prior scalar instruction duplication techniques as evaluated.

The proposed technique can significantly benefit from architectural support. A single additional SIMD instruction that compares two elements of an SIMD register and generates an exception when they are not equal would reduce error checking code overhead to a single instruction. This will make it feasible to check each duplicated instruction and perform immediate error correction.

Modern computing systems use cryptographic algorithms to secure access and data being processed. Malicious attackers may attempt to break these system to extract critical information through injecting errors. We extended our error detection framework to cover fault attacks on cryptographic algorithms in Chapter 5. Leveraging vector resources, our approach has potentially better cost performance ratio, and can be applied to an off-the-shelf processor. Fault injection was performed with the Intel Pin tool and showed that almost all the injected faults were detected by our technique with reasonable performance overhead.

7.2 Future Work

The techniques proposed in this dissertation may inspire new research on fault tolerance and fault attack countermeasures, and some potential directions are listed as follows:

- **Implementation on other processors with vector units.** The proposed SIMD-based error detection and fault attack countermeasure were implemented using LLVM on Intel processors with vector units. It will be also interesting to explore other platforms including embedded processor (e.g. ARM) and GPU as these processors are also equipped with vector units. For example, Wadden et. al [108] proposed a compiler
assisted software RMT on GPU. They mainly focused on converting GPGPU kernels into redundantly threaded versions. In addition, there are tools available to inject errors to GPGPU's register files and compute the Architectural Vulnerability Factor (AVF) [53, 34]. However, to the best of my knowledge, no instruction duplication has been experimented on GPGPUs. It might be worthy to investigate.

- **Hardware implementation of the proposed technique.** The current SIMD-based error detection technique still causes high performance and code size overhead due to the inefficiency in performing some operations, e.g. gather/scatter, error checking, and vector division, etc. One potential possible solution to further reduction of the cost is implementation of proposed approach in hardware. For example, one can design a specific vector unit to do the error checking whenever it is signaled to do so.

- **Application on side-channel attack mitigation.** The SIMD-based approach showed effectiveness on mitigating fault attacks. Another interesting topic to investigate is the mitigation of side channel attacks since countermeasures against fault attacks and side channel attacks are conventionally different. For example, countermeasures against side channel attacks are often based on masking and hiding, while countermeasures on the former frequently rely on redundancy. The proposed approach might be altered to provide both masking and redundancy. That said, we might be able to modify this technique to prevent computing systems from both attacks.

- **Evaluation on multithreading benchmarks.** The current work only concentrated on the single threaded programs. Although we believe that it should be directly applicable to multi-threaded programs. It would be interesting to evaluate the effectiveness.
Bibliography


