2017 Workshop on Fault Diagnosis and Tolerance in Cryptography

CAMFAS: A Compiler Approach to Mitigate Fault Attacks via Enhanced SIMDization

Zhi Chen*, Junjie Shen*, Alex Nicolau*, Alex Veidenbaum*, Nahid Farhady Ghalaty‡ and Rosario Cammarota†
† Qualcomm Research, San Diego, USA,
* University of California Irvine, Irvine, USA
‡ Accenture Cyber Security Technology Labs, Virginia, USA
Email: {zhic2, junjies1, nicolau, alexv}@ics.uci.edu, nahid.farhady@accenture.com, ro.c@qti.qualcomm.com

***Abstract*—The trend of supporting wide vector units in general purpose microprocessors suggests opportunities for developing a new and elegant compilation approach to mitigate the impact of faults to cryptographic implementations, which we present in this work. We propose a compilation flow, CAMFAS, to automatically and selectively introduce vectorization in a cryptographic library - to translate a vanilla library into a library with vectorized code that is resistant to glitches. Unlike in traditional vectorization, the proposed compilation flow uses the extent of the vectors to introduce spatial redundancy in the intermediate computations. By doing so, without significantly increasing code size and execution time, the compilation flow provides sufficient redundancy in the data to detect errors in the intermediate values of the computation. Experimental results show that the proposed approach only generates an average of 26% more dynamic instructions over a series of asymmetric cryptographic algorithms in the Libgcrypt library.**

I. Introduction

Instruction duplication and triplication countermeasures (herein after "instruction redundancy") [1] 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 [2], 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 [1], instruction duplication can cause up to `3.4x` times performance slowdown.

Yuce et al. [3] 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 paper 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 work 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 "`SIMD`ization"), 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 [4]. Similarly to the state-of-the-art time-based instruction duplication techniques, e.g., [1], our `SIMD`ization based countermeasure doesn't need to be aware

[1]https://software.intel.com/en-us/articles/improving-openssl-performance

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 circumstance 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. [1] 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 work 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 [1], which is vulnerable to the single fault injected at certain pipeline stages [3].

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.

In our experiments, we have applied CAMFAS to cryptographic algorithms in the Libgcrypt library.[2] 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.

II. Fault Model

This work considers an attacker capable of injecting glitches, as in [5], [6], which could corrupt data and/or instruction (mainly instruction skipping). 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 at different elements in a 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 paper because it is much more difficult to inject these type of faults [1].

An attacker can skip an instruction or replace it with an effective 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 [7], instructions are rearranged so that the vulnerable branch instruction is followed by a *default-fail* error handling module.

[2] https://www.gnupg.org/software/libgcrypt/index.html

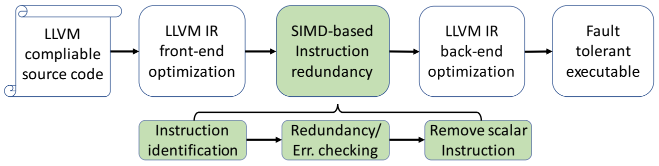
LLVM compliable source code
LLVM IR front-end optimization
SIMD-based Instruction redundancy
LLVM IR back-end optimization
Fault tolerant executable
Instruction identification
Redundancy/ Err. checking
Remove scalar Instruction

Fig. 1: CAMFAS compilation flow for introducing SIMDization based approach for fault detection in cryptography.

III. Compilation flow for fault detection in cryptography

Figure 1 depicts the compilation flow implementing CAMFAS for fault tolerant code generation in cryptography. The instruction redundancy pass in Figure 1 handles the Intermediate Representation IR files in three steps: (a) instruction identification and redundancy; (b) error checking code insertion; and (c) scalar code deletion. The transformed IR files are then passed to the LLVM back-end for further optimizations and register allocation before generating the fault tolerant executable.

Note that, although the countermeasures in the rest of the paper will only focus on fault detection, CAMFAS can be extended to support fault correction by triplicating the data in a SIMD register and performing majority voting among the resultant values. The correction can be implemented as a standalone module invoked on fault detection.

The compilation framework is expected to be applicable to all LLVM supported architectures with SIMD instruction set extension (e.g., Intel, AMD, ARM-based, etc.), as it focuses on the IR form of the source files which is normally platform independent. For illustrative purpose, this work illustrates an application of CAMFAS to Intel SSE and AVX/2 SIMD instruction set extensions, and to the case of instruction duplication.

A. Application of the framework to Instruction Duplication

In this section we present how CAMFAS handles vector based instruction duplication. The first step of our implemented `LLVM` pass is to traverse each individual `IR` instruction. This step determines if it is necessary to duplicate a discovered instruction during the traversal. If so, a vector instruction that contains two exact copies of the data used by the original instruction will be inserted to replace the original one. The framework copes with different instruction categories as follows:

- **Arithmetic, logic, and shift/rotate instructions:** These `IR` instructions are directly duplicated via an equivalent vector instruction and the result is saved in a vector intermediate register. Figure 2 shows the transformation of a scalar arithmetic operation (on the top) to its alternative `SIMD` form (at the bottom). `A′` is the replica of data `A` that can be either loaded from the memory or computed by preceding instructions.

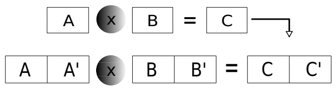

Fig. 2: Duplicate and vectorize scalar arithmetic isnstructions.

- **Memory instructions:** `CAMFAS` protects all load and store addresses by using gather and scatter instructions to duplicate memory access computation. A gather instruction effectively eliminates the extra load used in the previous scalar instruction duplication. Researchers in [3] 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. However, our duplication avoids executing identical loads sequentially. It 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 public key (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(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. 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 μops than a `mov` instruction. To avoid the prohibitively expensive cost, we only protect memory addresses and gather the value from (e.g., `A`) the same memory location into a SIMD register in this paper. Duplication and verification of memory contents are left as future work.

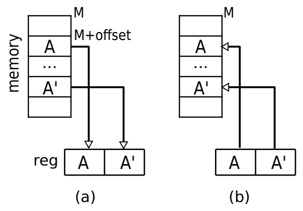

Fig. 3: Memory instruction duplication using (a) gather, (b) scatter.

- **Branch instructions:** All comparison instructions are duplicated since their SIMD alternatives are supported by `SIMD` instruction extensions, thus branch conditions are also protected. Attacks on branch instructions that change the branch target are not counteracted directly in our approach due to three reasons. First, existing signature-based techniques, e.g. [8], [9], can be employed to guard the integrity of control flows. Second, data integrity faults are more important in cryptographic applications since crypto algorithms usually consist of `xor` and `shift` instructions. Third, fault attacks mostly exploit the corrupted data result (DFA) rather than the control flow. Our goal is to defeat fault attacks, hence it is more important to consider data integrity instead of control flow integrity.
- **Function calls:** Function calls are tackled differently depending on if it is a subroutine call or a library call. Subroutine calls are not duplicated since instructions in the callee will have their own replicates. The majority of mathematic library calls are duplicated using their vector forms since `LLVM` has an array of vector compatible intrinsics to support math library functions. We can conveniently replace a scalar library call with its vector counterpart by extending the size of the input data types, and the LLVM back-end will generate the vectorized library call.
- **Other instructions:** Other instructions such as stack manipulation instructions (e.g., push and pop) are not duplicated because of two major reasons. First, these instructions are not present in the `IR` form. Second, no equivalent `SIMD` instruction exists to achieve the same purpose. Conversion instructions are secured because `LLVM IR` provides us the vector version of truncation, type conversion, and sign extension instructions.

B. Error Checking Code Insertion

Once an instruction is duplicated and vectorized, we need to decide if a check is required to be inserted for fault detection. Checks can be added to various code places to trade off the performance/code size and the fault coverage. For example, a check can be inserted immediately after a vectorized instruction to compare equality of the upper and bottom 64 bits of a resultant SIMD register. While it attains full coverage of the attacks since all duplicated instructions are validated, checking all duplicated instructions may overreact to fault detection as many checks are actually superfluous. This is because the value change in a register will generally propagate to influence all the registers that are directly or indirectly dependent on this register. In some cases, this change is masked by some operations on the execution path, i.e., a shift operation could mask all the changed bits in a register that will be shifted out.

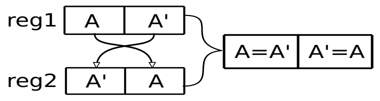

Fig. 4: Error checking operation

Therefore, to leverage the fault coverage rate at the expense of a reasonable performance and extra instruction overhead, fault checking will be only performed at certain program points, such as before stores, function calls, conditional branches, etc.

Figure 4 demonstrates how a comparison is performed to check the equivalence of the two values stored in the upper and lower half parts of a SIMD register, reg1. First of all, a shuffle has to be executed to swap these two values in reg1 since there is no horizontal comparison instruction to achieve this purpose. The shuffled result is stored in another SIMD register, say reg2. A vector comparison is then run to check the equivalence of the values in the corresponding lanes of reg1 and reg2. In other words, the fault checking requires at least four SIMD assembly instructions in the back-end, e.g., vpshufd, vpcmpeqq, vptest, and jne. These extra instructions will penalize the performance and instruction count. Fortunately, checks are only inserted infrequently. We can thus expect a reasonably low overhead.

IV. Evaluation

This section provides an experimental evaluation of 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 *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.

A. 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]

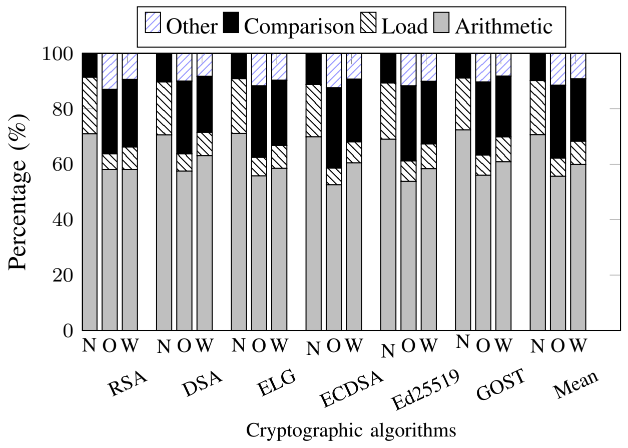

Fig. 5: The breakdown of faults injected on each type of instruction for different mechanisms: no fault detection (N), duplication without gather and scatter instructions (O), duplication with gather and scatter instructions (W).

B. 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 [10] 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.

[3]https://www.riscure.com/security-tools/hardware/vc-glitcher

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.

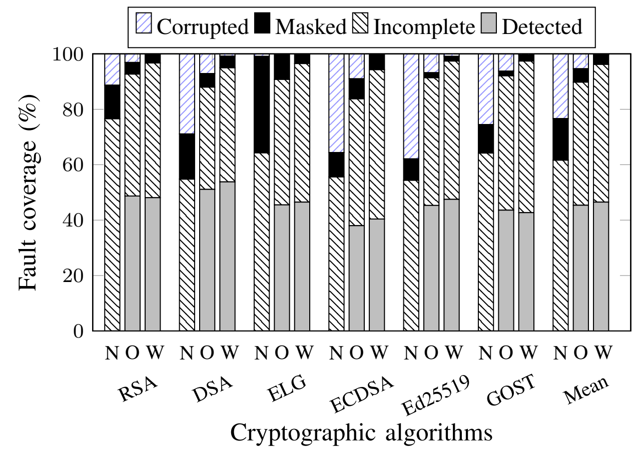

Fig. 6: Fault coverage by each cryptographic algorithm among different mechanisms: no fault detection (N), duplication without gather and scatter instructions (O), duplication with gather and scatter instructions (W).

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 executes at least 5 times longer than producing the masked results.

Figure 5 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 Section III. Note that the last category doesn't apply to the baseline, since no such operation exists in the original program.

Figure 5 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 6 associated with Table I shows the effectiveness of our vector based error detection approach in detecting injected faults. As it is tabulated in Table I, `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).

Approach	Detected	Incomplete	Masked	Corrupted
N	0	61.65%	15%	23.35%
O	45.37%	44.43%	5.83%	5.37%
W	46.5%	49.72%	3.42%	0.36%

TABLE I: 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 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.

C. 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 crytposystems 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 6, the number of "Incorrect" outputs has reduced from 23.35% to 0.36%.
- Differential Fault Intensity Analysis (DFIA) [6]: 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: [3] 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 II, 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.

D. 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.

Performance overhead. Figure 7 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 7, 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 `divq/mulxq` instructions are executed sequentially to obtain the upper and bottom quadword values. These two values are packed into a SIMD register using `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

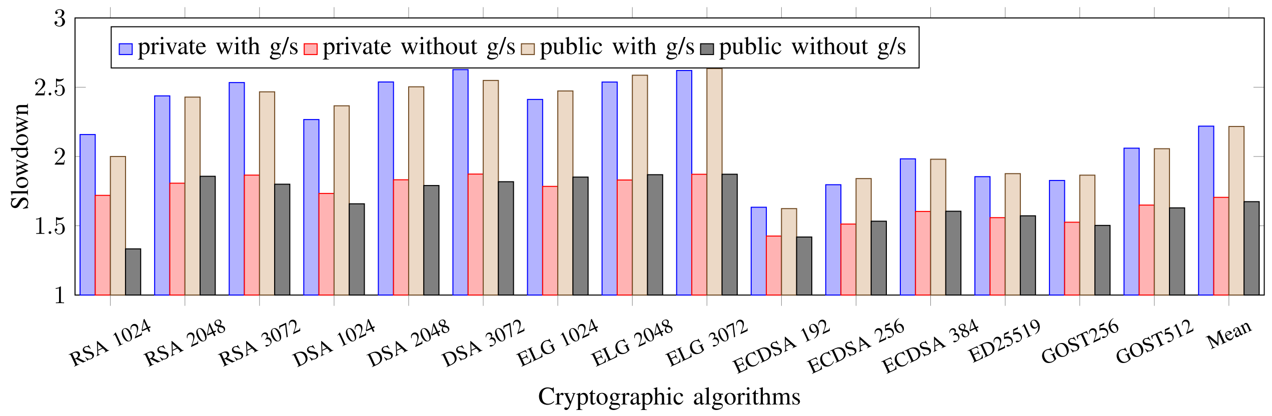

Fig. 7: The slowdowns of error detection enabled cryptographic algorithms to compute private and public keys compared to the original ones without error detection for different input sizes.

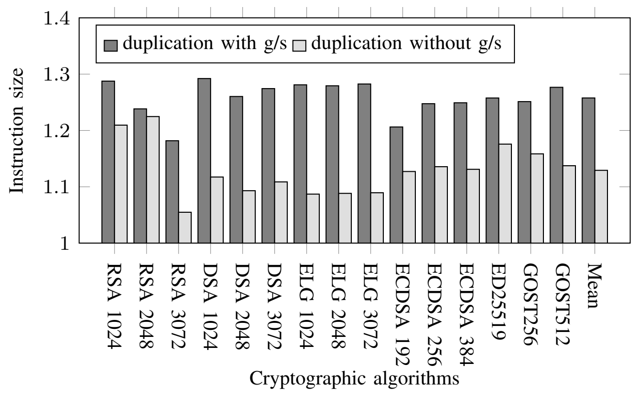

Fig. 8: The dynamic instruction count overhead for duplication with and without gather/scatter instructions.

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 [11].

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., `vpgatherqq` instruction needs 18 and 15 clock cycles on Skylake and Broadwell [12], respectively. At least the same latency would be expected on the Knights Landing processor because it needs 6 micro operations (μ`ops`) vs 5 on Skylake [11].

Dynamic instruction count overhead. Figure 8 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., `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 7. 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 8 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.

V. Related Work

Recent prior art focused on specific hardware approaches to implement fault detection and mitigation [13], [5]. 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. Prior art on software based countermeasures propose mitigations either at the algorithm level [14], [15] or at the instruction level [1], [2] 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 [3].

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 [1], 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 [16], but they didn't address the fault tolerance problem. Chen et at. proposed a compilation framework to utilize `SIMD` resources for fault tolerance [17]. 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.

VI. Conclusion

This work proposes `CAMFAS`, a new compiler flow for fault tolerance in cryptography using vectorization. The proposed technique exploits the underutilized vector resources on processors for redundancy purposes. `CAMFAS` was implemented in the `LLVM` compiler infrastructure. It automates the insertion of instruction redundancy in cryptographic code. In addition, the compiler oriented countermeasure converted the traditional temporal redundancy based instruction duplication into spatial redundancy with relatively low performance and code size overhead. The experimental result showed that our proposed approach only required up to an average of 26% more dynamic instructions compared to the originally unprotected cryptographic algorithms, but it achieved almost full fault coverage.

References

[1] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures Against Fault Attacks on Software Implemented AES: Effectiveness and Cost," in *WESS*, 2010, pp. 7:1–7:10.

[2] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont, "Lightweight fault attack resistance in software using intra-instruction redundancy," Cryptology ePrint Archive, Report 2016/850, 2016.

[3] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software fault resistance is futile: Effective single-glitch attacks," in *FDTC*, 2016, pp. 47–58.

[4] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2004, pp. 75–86.

[5] B. Yuce, N. F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. Schaumont, "FAME: Fault-attack Aware Microprocessor Extensions for Hardware Fault Detection and Software Fault Response," in *HASP*, 2016, pp. 8:1–8:8.

[6] N. F. Ghalaty, B. Yuce, and P. Schaumont, "Analyzing the efficiency of biased-fault based attacks," *IEEE Embedded Systems Letters*, vol. 8, no. 2, pp. 33–36, 2016.

[7] S. Endo, N. Homma, Y.-i. Hayashi, J. Takahashi, H. Fuji, and T. Aoki, "A multiple-fault injection attack by adaptive timing control under black-box conditions and a countermeasure," in *COSADE*, 2014, pp. 214–228.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ccs*, 2005, pp. 340–353.

[9] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *CGO*, pp. 243–254.

[10] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education," in *WCAE*, 2004.

[11] A. Fog, "Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," http://www.agner.org/optimize/instruction_tables.pdf, 2017.

[12] "Intel Intrinsics Guide," https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[13] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[14] M. Medwed and J.-M. Schmidt, "A generic fault countermeasure providing data and program flow integrity," in *FDTC*, 2008, pp. 68–73.

[15] R. Karri, G. Kuznetsov, and M. Goessel, "Parity-based concurrent error detection of substitution-permutation network block ciphers," in *CHES*, 2003, pp. 113–124.

[16] K. C. Pabbuleti, D. H. Mane, A. Desai, C. Albert, and P. Schaumont, "SIMD acceleration of modular arithmetic on contemporary embedded platforms," in *HPEC*, 2013, pp. 1–6.

[17] Z. Chen, A. Nicolau, and A. V. Veidenbaum, "SIMD-based Soft Error Detection," in *CF*, 2016, pp. 45–54.