SECURITY IMPLEMENTATIONS IN THE POLYHEDRAL COMPILATION INFRASTRUCTURE

Kerri McMahon, Computer Science, St. Joseph's College, Patchogue, New York, 11772

ABSTRACT

Fault attacks are being studied more and more due to their increasing efficiency in breaking cryptographic implementations. The intent of this type of attack is to extract sensitive information about cryptographic systems, which is a problem with the demand for high performance computing. A countermeasure to this attack is instruction duplication. This defense mechanism modifies instructions to incorporate redundancy in the intermediate representation stage of compilation. The polyhedral model is a popular area of study in high performance computing. At the same time, there lacks study in making sure such advancements still enable compilers that use the polyhedral model to remain secure. In this work, we investigate how PoCC (Polyhedral Compiler Collection), a polyhedral compilation software, performs with security implementations aimed to detect a fault attack.

INTRODUCTION

The polyhedral model represents source instructions as theoretical polyhedra for optimization. **The advantage of this is that the complexity of the instructions is based on the complexity of the structure itself, and not the number of elements represented.** [1] The polyhedral model performs based on statement iterations, and these iterations can be represented as matrices. The range of statement instances make up an instruction's iteration domain. During the compilation process, these instructions get represented as a PAST (Polyhedral Abstract Syntax Tree). While traversing and manipulating the PAST to incorporate fault detection, it is important to ensure that the scheduling tree preserves its desired logic as well as peak code optimization. Our method detects errors through frequent consistency checks, as well as provide user specification for duplication.

We experimented with PoCC to observe the compiler's performance with fault error-detection implementation. Fault attacks can be hardware or software based. We focus on software, implementing duplication and consistency checks at compile time. The PAST is manipulated directly in between the front-end and back-end optimization processes. The main focus of this work is to modify the compiler infrastructure to handle n-many syntax trees rather than one single structure for the purpose of consistency checking.

METHODS

The API source code for operations on the PAST was modified directly, and the compiler framework as a whole was altered to handle the processing of *n* many syntax trees, modify the duplicated instructions to rename any written array-references for the logical consistency of the program, and perform error detection before the back end optimizations. During the passing of the scheduling structure from front-end to back-end optimizations, an array of syntax tree deep copies are created. The original tree is optimized and its code is generated normally as the clones are traversed and modified.

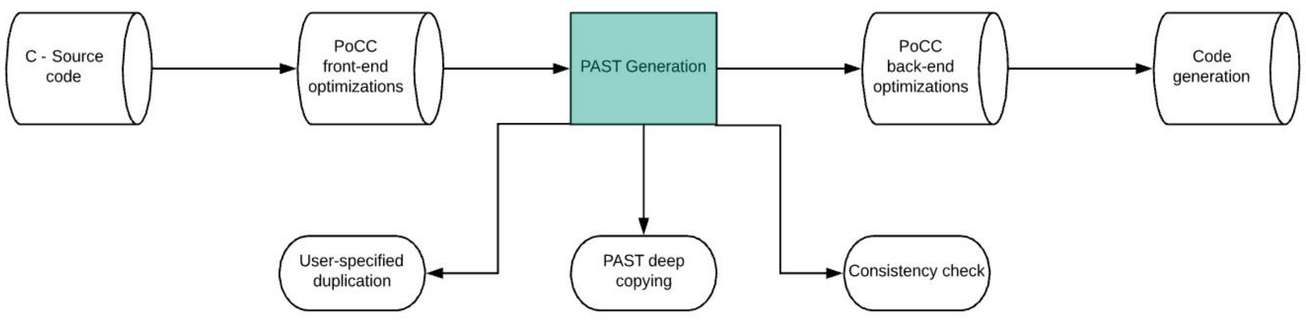


Figure 1: General workflow of modified compilation process.

RESULTS

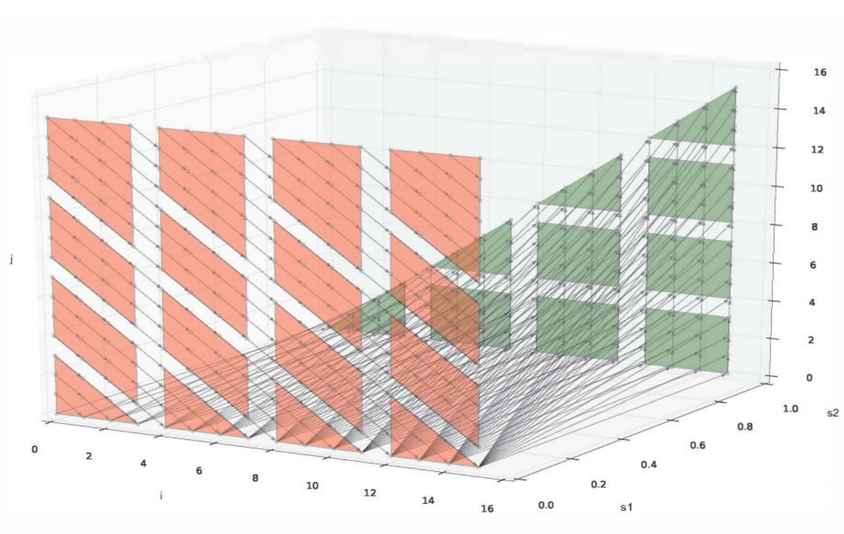
Tests were performed on an Intel Core i5 at 1.8GHz with 8 GB of memory (2 cores), with L2 Cache (per core) at 256 KB and L3 Cache at 3 MB. We experimented from a mixed set of stencil kernels, linear algebra kernels, and applications. The showed results are execution times in seconds for the original PoCC workflow compared to execution times with fault-attack detection implementations. Tests were run on various PolyBench/C benchmarks, and recorded times are the average compilation times over ten trials of the original PoCC compilation compared to PoCC with fault attack detection modifications.

```
#pragma scop
    for (i=0; i<N; i++)
      for (j=0; j<N; j++) {
    C[i][j] = C[i][j] * alpha;
    for (k = 0; k < N; ++k)
      C[i][j] += beta * A[i][k] * B[k][j];
      }
#pragma endscop
```

```
 #pragma scop
 if (N >= 1) {
   for (c1 = 0; c1 <= (N + -1); c1++) {
     {
 for (c3 = 0; c3 <= (N + -1); c3++) {
   {
 C[c1][c3] = C[c1][c3] * alpha;
 for (c5 = 0; c5 <= (N + -1); c5++) {
   {
 C[c1][c3] += beta * A[c1][c5] * B[c5][c3];
   }
 }
   }
 }
     }
   }
 }
 #pragma endscop
```

```
  register int c1, c3, c5;
#pragma scop
if (N >= 1) {
  for (c1 = 0; c1 <= (N + -1); c1++) {
    {
for (c3 = 0; c3 <= (N + -1); c3++) {
  {
C[c1][c3] = C[c1][c3] * alpha;
for (c5 = 0; c5 <= (N + -1); c5++) {
  {
C[c1][c3] += beta * A[c1][c5] * B[c5][c3];
  }
}
  }
}
    }
  }
}
#pragma endscop
```

Figure 4 - dgemm.c kernel code comparison




```
for(i=0; i<=15; i++){
 for(j=0; j<=15; j++)
  a[i][j] = a[i+1][j-1];
 for(j=0; j<=i; j++)
  b[i][j] = b[i][j+1]+a[i][0]; }
```

Figure 2 - Nested for loop's tiled polyhedron and dependencies [2]

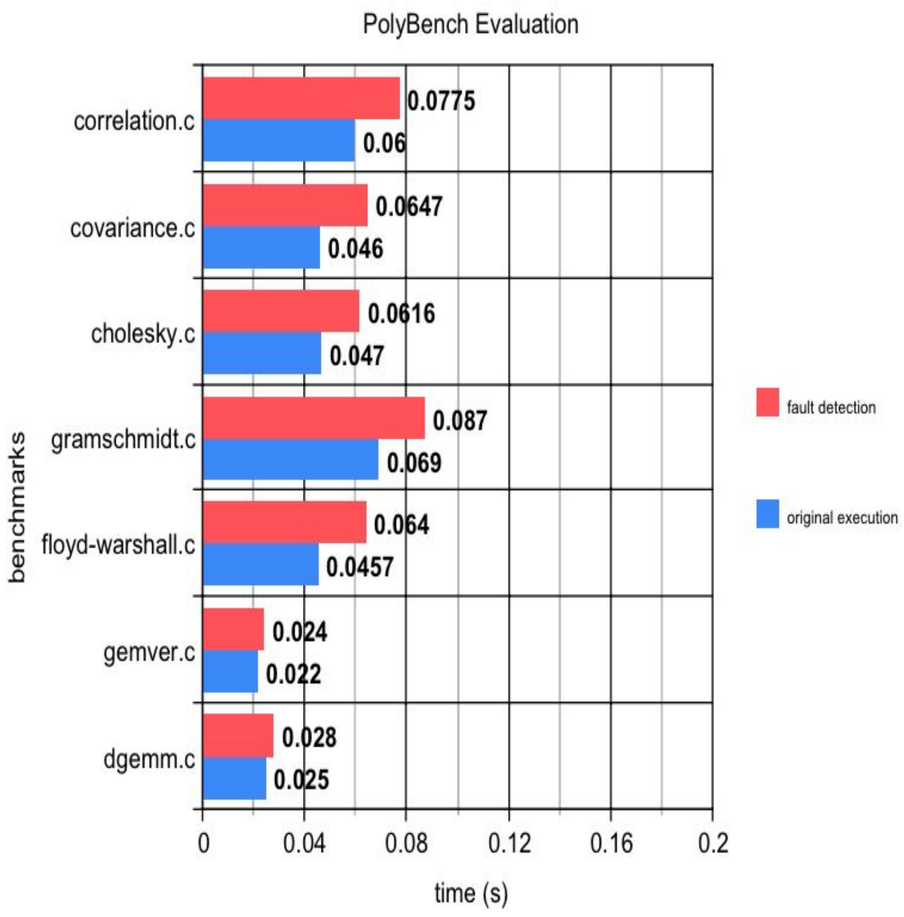


Figure 3 - PolyBench performance evaluation

DISCUSSION & FUTURE WORK

Fault-attack detection in the polyhedral model still allows the compiler to perform close to as desired. This work shows the beauty of the polyhedral model; flexibility. Further work will include investigation of PoCC's behavior with other attack defense implementations at this granularity as well as investigate how to further minimize execution time in addition to the efforts already made.

REFERENCES

[1] "Polyhedral.info." *Polyhedral Compilation*, MIT, 2014, polyhedral.info/.

[2] M. Palkowski and W. Bielecki, "An iteration space visualizer for polyhedral loop transformations in numerical programming," *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Gdansk, 2016, pp. 705-708.

ACKNOWLEDGEMENTS

I would like to thank those in the Computational Science Initiative department for their hospitality and support throughout my time with them. This project was supported in part by the U.S. Department of Energy, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internships Program (SULI).