Silent Data Corruption Detection with Program Synthesis

Kerry Tu*
Weixin Yu*
kjwtu@stanford.edu
weixinyu@stanford.edu
Stanford University
Stanford, California, USA

Abstract

Reliability issues in hardware circuits have become increasingly important in the age of data and computation explosion. Hyperscalers are reporting Silent Data Corruptions (SDCs), computation errors where the only symptom is an incorrect result. Those errors are extremely difficult to locate and debug. However, hardware design and manufacturing efforts can only reduce the possibility of defects. Existing software-level proactive testing approaches either take a long execution time or focus on only part of the sources of hardware defects.

We present our automated tool, packaged as an LLVM compiler pass, that generates alternative but functionally equivalent instruction sequences using Program Synthesis techniques. We target individual instructions at the LLVM-IR level as the source of transformation. The compiler pass transforms a valid LLVM program into a functional consistency check that enables SDC detection. The tool is able to localize the specific LLVM instruction where silent data corruption occurs.

We also implemented an error injection and detection framework at the LLVM-IR level to evaluate the effectiveness of our approach. The errors are inserted on top of the 6 instrumented tests generated from C programs at the LLVM-IR level. Our results demonstrate the effectiveness of our approach in detecting SDCs with acceptable performance overhead (within 3%). The instrumentation time for a program with around 700 lines of code is around 1 minute.

*Both authors contributed equally to this research.

CS357S '25, Stanford, CA

ACM ISBN 357-3-5735-7357-3/2025/03
https://doi.org/3573573.5735735

CCS Concepts: • **Computer systems organization** → **Reliability**; **Redundancy**; • **Computing methodologies** → *Equation and inequality solving algorithms.*

Keywords: Silent Data Corruptions, Program Synthesis, Rosette, Functional Consistency, LLVM, Error Injection, Instruction Duplication

ACM Reference Format:
Kerry Tu and Weixin Yu. 2025. Silent Data Corruption Detection with Program Synthesis. In *Proceedings of CS357S Conference (CS357S '25).* ACM, New York, NY, USA, 10 pages. https://doi.org/3573573.5735735

1 Introduction

Silent Data Corruptions (SDCs) are computation errors in which the only symptom is an incorrect result. Hyperscalers are reporting SDCs. For example, Meta recently reported cases where "a simple computation like 1.1^{53}" resulted in the wrong answer and missing rows within the database [3, 8]. As the incorrect result is the only symptom, the error can only be detected after accessing the database containing that row. Therefore, it is very difficult to know when, where, or how the corruption occurs.

The inherent causes of the problem are hardware vulnerabilities and the assumption from the software level that the hardware works as expected. At the scale of hyperscalers, very low hardware defect rates for single machines accumulate to produce inevitable errors.

Current SDC mitigation solutions by both datacenter operators and researchers employ the common idea of proactive testing. Google's SiliFuzz [13] repeatedly tests cores using randomly generated tests. To reduce the test size, Vega [10] leverages knowledge from the hardware and generates targeted tests focused on transistor aging. ITHICA [9] employs an intra-thread instruction checking technique by inserting duplicate instructions and performing consistency checks. However, the method might not catch all hardware defects. Specifically, it does not perform well in the case where hardware defects behave consistently for particular inputs.

Building on top of ITHICA, we inherit the intra-thread instruction checking and take a different approach. Instead of duplicating each instruction, we generate alternative instruction sequences that perform the same architectural function

from the original instruction using program synthesis. In our implementation, we explicitly avoid reusing the arithmetic operator from the original instruction for our newly generated sequence. As a result, our approach works even if defect-induced SDCs manifest consistently.

In particular, we generate replacement code snippets at the LLVM-IR level using Rosette, a program synthesis tool. Our framework performs program synthesis and instruments the code at LLVM-IR level and is thus directly applicable to most architectures. The tool is essentially an LLVM compiler pass that instruments any valid LLVM code. Moreover, our implementation enables users to tune the number of alternative snippets and the maximum length of generated alternative instruction snippets. We focus our work on arithmetic operations at the LLVM-IR level, due to their inherent compatibility with Rosette program synthesis.

Moreover, we implemented an error injection and detection framework at the LLVM level to evaluate the effectiveness of our tool. On each execution of the modified program, a bit manipulation is introduced at a certain probability (tunable) to emulate SDC-manifesting hardware. The framework is able to pinpoint the line number of the original instruction in case a corruption occurs by exiting with a nonzero return code indicating the position of failure.

To evaluate the effectiveness of alternative code snippet insertion and functional consistency check on early detection of SDCs, we run our framework on a test suite consisting of six C programs that implement common algorithms and data structures, including priority queue and backtracking.

In our experiments, the generation time is around one minute for an original program with around 700 lines of code, and the execution overhead of the instrumented program is at most 3%. Under our error injection framework, we also demonstrated the effectiveness of our tool in detecting SDCs.

Overall, we make the following contributions:

- We present our framework and tool that generates and inserts alternative but functional equivalent code snippets back into original code at the LLVM-IR level. We implement the tool as an LLVM compiler pass.
- We implemented an error injection and evaluation framework at the LLVM-IR level to model the behavior of defected hardware. The evaluation framework is able to pinpoint the line of code where the corruption occurred, resolving the inherent difficulty of locating errors for SDCs.
- We demonstrate the effectiveness of our approach with 6 C programs. We show that the overhead of both program synthesis and execution is acceptable.

2 Background

2.1 Hardware Defects

Hardware failures can be classified by source, including design bugs, permanent faults, and transient faults.

Systematic Faults. Design bugs are due to erroneous design, where bugs occur consistently on most chips manufactured according to the design. There has been research on using formal verification techniques to prove the correctness of designs, including Quick Error Detection (QED) approaches [1], symbolic repair [11], and domain-specific languages [4].

Permanent Faults. Permanent faults are due to manufacturing defects. Processors with permanent faults are either born with the fault, or they develop erroneous behavior due to aging-related issues. In particular, Vega focuses on transistor-aging-related silent data corruption detection [10]. Even though these type of errors are permanent, they typically occur under a specific execution context. Thus, due to the complexity of modern processor design, assembly instructions with the exact same registers and operators may not actually be mapped to the same hardware execution unit. ITHICA [9], for example, builds on this insight and performs functional consistency checks on replicated instructions.

Soft Errors. Soft errors in processors are temporary faults that occur randomly and often disappear after a short time. The cause of those errors could be cosmic rays and radiation, power fluctuations, electromagnetic interference, and temperature variations.

2.2 Silent Data Corruption

Silent data corruptions (SDCs) are undetected errors generated in computing, caused by hardware defects which are not consistent over inputs, program execution, or the processor's life cycle. Design bugs are not included in the category. Therefore, targeted mitigation techniques other than traditional silicon validation approaches like Quick Error Detection (QED) are necessary.

The only symptom of SDCs is incorrect data. Therefore, errors can propagate across the stack silently. Only after an unpredictable amount of time when the data are accessed again can the error be detected. This has made SDCs very hard and laborious to trace.

SDCs have had a negative impact on large-scale infrastructure services. For example, Meta reported errors that have caused data loss and cost months of debug engineering time [3].

The inherent cause of SDCs is the mismatching assumption made by software applications that processors always behave as expected with respect to their architectural specifications. While there is no existing technique to eliminate all hardware manufacturing defects yet, researchers aim to address this critical issue by proactively testing the underlying hardware at the software level.

2.3 Related Work

Previous research by Dixit [3] has surveyed various methods to debug SDCs, in both the hardware and software spaces.

Some recommended hardware mitigations include the use of protected datapaths, specialized screening, understanding at-scale behavior, and architectural priority. The paper also offered several options for software mitigations for SDCs, including redundancy and fault-tolerant libraries. In Meta's own research [2], they created two suites for testing hyperscalar systems: Fleetscanner and Ripple. Fleetscanner focused on out-of-production testing, in which the servers are in maintenance and not currently running productive workloads, while Ripple focused on in-production testing, which allowed testing without halting running systems. Furthermore, Fiala's RedMPI library [5] has been shown to be effective in detecting and correcting soft errors in MPI message passing applications, using a mix of redundancy and consistency protocols. Hari's work [7] attempts to improve the efficiency of SDC detection systems by analyzing what types of code sections caused the most SDCs (90% of SDCs were caused by 5.4% of static instructions). With this analysis, they successfully improved efficiency by placing detectors at these fault-prone code sections (comparing similar computations, checking value equality, performing range checks, and performing mathematical tests), compared to a purely redundancy-based system.

2.4 Motivation

2.4.1 ITHICA. ITHICA performs intra-thread consistency checking for detecting defect-induced SDCs [9]. Their work is based on the observation that SDCs tend to manifest inconsistently. However, as clearly indicated, this approach is unable to catch defects that occur consistently for specific inputs.

The idea of consistency checking proved to be simple, effective, and independent of complete architectural specifications. To employ the idea while mitigating the problem mentioned above, we propose our approach and tool that generate functionally equivalent but different instruction sequence to insert into the test program. By using different operators and constants (depending on situations), our tool successfully avoids the inconsistency assumption on hardware defects.

2.4.2 Program Synthesis. Program synthesis tools have become powerful and readily applicable to generate programs given simple specifications. With the inherent support of bit vectors in Rosette, the program synthesis framework we use, we are able to model and synthesize all LLVM arithmetic instructions on the fly, generating instruction sequences with desired behavior.

In cases where no alternative instructions are found by the program synthesis tool, we simply replicate the instruction, mirroring what ITHICA did. This ensures that we can handle those instructions as effectively as ITHICA does.

3 Approach

We design, implement, and evaluate a framework that helps detect silent data corruptions (SDCs). Figure 1 provides the design overview for the framework. It takes in a valid program in LLVM-IR, generates functionally equivalent instructions for each arithmetic operations, instruments the program, and executes repeatedly on a server to detect SDCs.

3.1 Alternative Instruction Generation

The existing approach has been to insert duplicated and comparison instructions into the program as a functional consistency checker. However, due to the nature of hardware execution, it is likely that duplicated instructions will undergo the same control path and be executed by the same functional unit in the underlying hardware architecture. If this happens to be true, the result of a duplicate instruction will always be equal to the original, defeating the purpose of being a consistency checker. On the other hand, current program synthesis tools have become powerful enough to generate programs given simple specifications. Therefore, we decided to employ program synthesis to generate functionally equivalent alternative instructions in place of duplicate instructions.

We perform the program synthesis at per-instruction granularity, while allowing the generated instruction sequence to contain multiple instructions (defined with a tunable parameter in our tool). Allowing longer instruction sequences usually increases the required time to perform the program synthesis. In other words, there are at most two source registers and one destination register in both the original and modified programs. For a single original instruction, Rosette program synthesis gives a recursive expression representing the generated sequence. Another translation we did was to translate the expression into a list of LLVM-IR instruction sequence.

3.2 Program Instrumentation

To achieve proper silent data corruption detection, the code created using program synthesis must be inserted back into the original Clang-generated LLVM code, and the result of the synthesized code must be equal to the result of the original line. If they match, then there is no detected silent data corruption. However, if they differ, the program detects an SDC and must halt execution. Ideally, the specific line from which the SDC originated should be able to be pinpointed, so that localized debugging or analysis may be performed if desired, post-SDC detection. Some language conversions are necessary throughout the process, as the input is a C program and the output is modified LLVM code or executable. The Rosette program synthesis implementation requires input in the Racket language and also outputs synthesized code in Racket.

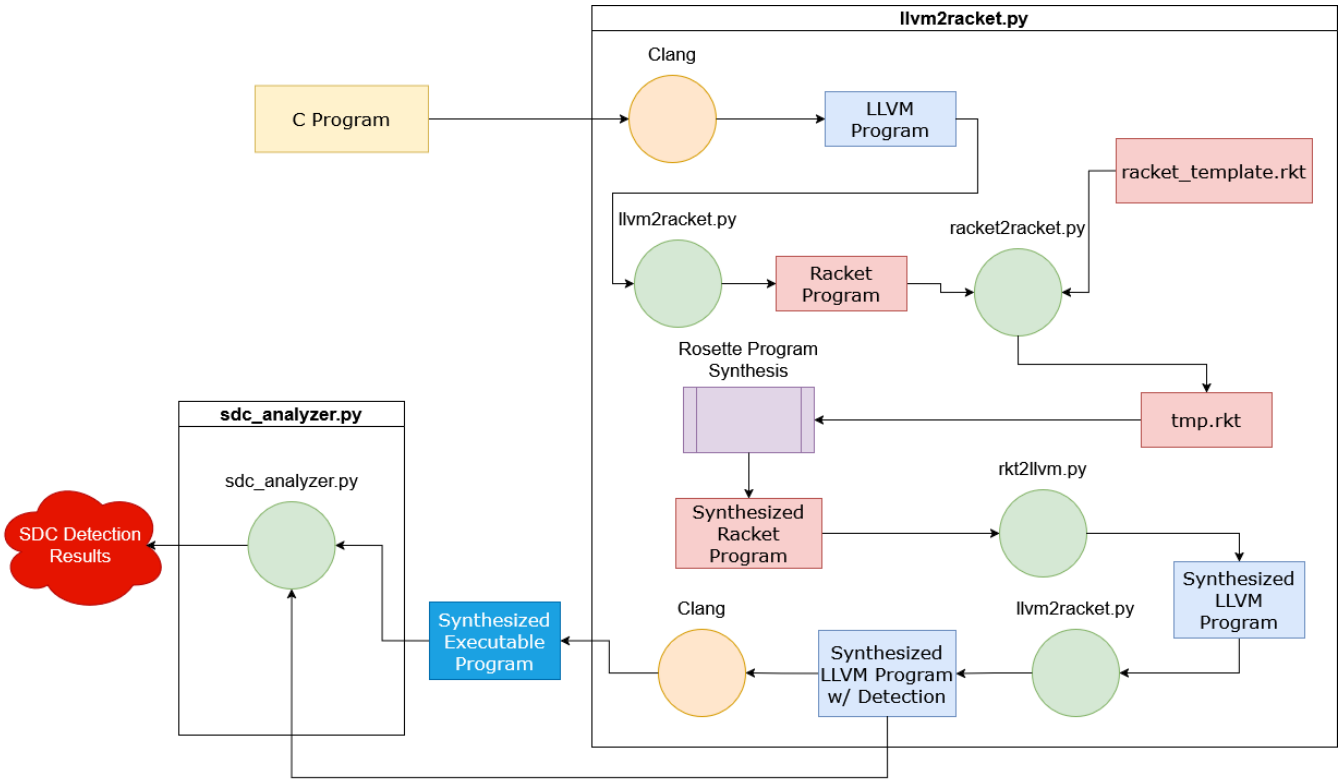

Figure 1. Toolflow for SDC Detection with Program Synthesis

4 Implementation

4.1 Alternative Instruction Generation

In our implementation, we used Rosette, a solver-aided programming language that extends Racket, for program synthesis. Rosette has built-in support for bit vectors, resulting in a direct mapping between LLVM-IR integer operations and Rosette. We take advantage of this feature, greatly simplifying the LLVM to Racket compiler pass.

We implemented the program synthesis following the standard guide of Rosette: constructing a grammar template with a specific depth, creating the specification on-the-fly based on the LLVM-IR input, and finally synthesizing an alternate program with the usage of symbolic variables.

To avoid trivial synthesis result such as a simply duplicated instruction, alternative instructions are constrained to use sets of LLVM-IR binary operators disjoint with the original instruction. This is implemented by creating the grammar template on the fly, excluding operators already used in the original expression.

As both the grammar template and the specification depend on the specific input of the LLVM-IR instructions, we decided to generate the racket program dynamically using Jinja2, a templating engine. We decide not to create a lookup table from general LLVM instructions to alternative instructions in order to accommodate diverse translations stemming from constant parameters within instructions. For example, there is no solution for a general `mul i32 %reg1, %reg2`, but various solutions exist for `mul i32 %reg1, 2`, which is equivalent to `add i32 %reg1, %reg1` as well as `shl i32 %reg1, 1`.

Therefore, for each arithmetic instruction in the source LLVM-IR program, a temporary Racket program targeting the specific instruction will be generated and executed, where the output is read by another Python program for further processing.

To synthesize instructions, Rosette needs both a grammar template of the instruction we are generating and a specification. The grammar template is a recursive definition in approximate Backus-Naur form (BNF), and the specification is defined as the semantics of the original instruction.

As shown in Figure 2, for the instruction `mul i32 %reg, 2`, the `bvmul` operator is excluded from the grammar template to avoid generating duplicate instruction. The specification is defined as multiplying by 2 (in practice, in Rosette grammar), and the result of the synthesis is `add i32 %reg, %reg`.

The alternative code snippet generated by Rosette can contain multiple instructions, depending on the depth set with the grammar template. The depth in which we experimented is 2. We thus implemented a compiler pass using recursive functions that translates nested Racket expressions into a list of LLVM-IR instructions. In addition, the single

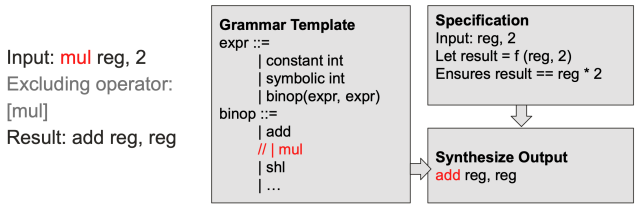

Figure 2. A simple program synthesis example.

static assignment rule in LLVM-IR is closely followed by having a `Counter` class variable whose member variable gets incremented whenever a new destination register's name is retrieved from the class.

It is reasonable to ask for as many different alternative instructions as possible, in order to exploit different functional units. Although there is no trivial way provided in Rosette, we partially tackle the problem by exploiting our dynamic code generation approach. Between Rosette program executions on the same LLVM instruction, we modify the grammar template for the generated program according to the result from each iteration. We maintain an *exclusion list* in Python, keeping track of all operators that have already been used in either the original instruction or the previously generated alternative instructions. The arithmetic operators listed in the Rosette grammar template for the next iteration are then calculated as the difference between the set of all arithmetic operators and the *exclusion list*. The exclusion would possibly prune programs that are both interesting and different from the previously generated ones, but it serves as an approximation.

Following the example in Figure 2, as shown in Figure 3, we add the operator used in the generated solution in Figure 2 to the exclusion list and exclude it from the grammar template while keeping the specification unchanged. The synthesized output on this run becomes `shl i32 %reg, 1`, yet another alternative instruction.

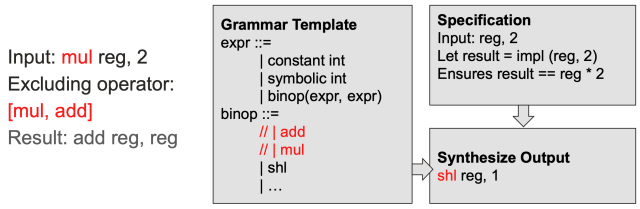

Figure 3. Program synthesis for generating more alternative instruction snippets.

4.2 Program Instrumentation

The toolflow of the SDC detection is shown in Figure 1, and is completely automated. There are two main scripts within the toolflow: llvm2racket.py and sdc_analyzer.py. The llvm2racket.py script takes in a C program and generates both the synthesized LLVM program with the SDC detection mechanisms, and the synthesized executable program. The outputs are then fed into sdc_analyzer.py for evaluation, detailed further in the Evaluation Methodology section. This section will mainly detail the workings of llvm2racket.py, while Evaluation Methodology will go into the specifics of sdc_analyzer.py.

We begin llvm2racket.py with a C program as input, and this will be the code that the user wants to evaluate or run on servers. Using Clang, the C program is compiled into an LLVM program, which is then converted with the script to a Racket program. It iterates line by line through the LLVM program and determines whether each line is an arithmetic instruction (add, sub, mul, etc.). For each arithmetic instruction in the program, it converts the instructions from LLVM to Racket using a mapping of instructions, along with the necessary syntax. Due to Racket's lack of specific numeric types, we convert LLVM's integer types (i1, i32, i64) into Racket bit vector types of appropriate length. The conversion from LLVM to Racket is necessary because our implementation using Rosette program synthesis requires a Racket input and is incompatible by default with LLVM. Then the script uses parsed data and another script, racket2racket.py, to fill the holes in a racket template for Rosette, from the racket_template.rkt, generating tmp.rkt.

This tmp.rkt is then run in conjunction with Rosette to generate synthesized Racket code that is functionally consistent with the original LLVM arithmetic operation, but using a different set of arithmetic operations. The synthesized code is then fed as an input into rkt2llvm.py, which converts the synthesized Racket code back into LLVM for insertion back to the original LLVM. This synthesized LLVM code is inserted before its corresponding original LLVM instruction, and to a different temporary result register. Additional comparison code is also appended after that compares the original LLVM instruction's result with the result of the synthesized code, using an sdc_check function. This function checks whether the input registers are equal, and exits the program and returns a particular error code corresponding to the error line if the input registers differ. Once this is done with all arithmetic instructions in the original code, the complete synthesized LLVM program with SDC detection is compiled into an executable using Clang. This is the final output of llvm2racket.py to sdc_analyzer.py.

To illustrate the toolflow more clearly, we show the toolflow outputs from a simple C test program in Figure 4. This program, pow2, calculates and prints the value of 2 to the power of p, where p is 4. Figure 5 shows the terminal output from running llvm2racket.py. As expected, the script detects two arithmetic operations, `%11 = shl i32 %10, 1` and `%14 = add nsw i32 %13, 1`. They are converted to Racket codes `%dest0 = (bvshl %10 (int 1))` and `%dest1 = (bvadd %13 (int 1))`, respectively. These are then used to populate

the Racket template that runs Rosette to generate alternative Racket instructions that are functionally consistent, shown next to "Alternate LLVM:" in Figure 5 after converting back to LLVM. We can reason that the shift left by 1 is functionally equivalent to adding the input twice, and subtracting -1 from a register and then shift right by 0 is equivalent to the original add 1. The script appends the synthesized code, along with its corresponding sdc_check function, to a modified LLVM files and then compiled with Clang to executables. Figure 6 better illustrates this insertion. At the end of the script, the executable is run once to verify that the code is functionally consistent with the original. If there are no errors, it will print out "Execution Successful" with a return code of 0.

```c
#include <stdio.h>
int main() {
    int p = 4;
    int result = 1;
    for (int i = 0; i < p; i++) {
        result = result << 1;
    }
    printf("Result: %d\n", result);
    return 0;
}
```

Figure 4. Simple C Test Program (pow2).

5 Evaluation Methodology

To evaluate the effectiveness of our methodology, we can either run on real servers or artificially introduce errors into the program during runtime at a set probability. The sdc_analyzer.py script takes in the synthesized LLVM and executable outputs from the llvm2racket.py script and outputs a set of SDC detection results, including the number of SDCs detected by the synthesized detection code and a bar graph displaying the amount of errors found on each line. The specific output and runtime behavior depend on the current setting of INTRODUCE_ERRORS variable. For execution and analysis on real servers, INTRODUCE_ERRORS should be set to False, and there will be no artificial error injections into the system, only real SDCs. There will also be no count for the number of true errors in the analysis, as the system does not know how many true SDCs occur, only the number it successfully detects. Without access to a real server, or if the user wishes to control the probability of SDCs occurring for the sake of evaluation, INTRODUCE_ERRORS should be set to True. This will artificially inject SDCs into the system through the following algorithm. Using the synthesized LLVM program with detection, the script will parse the LLVM file and generate a separate LLVM file and executable, each corresponding to an error in results in one of the program lines. This is done by adding the value 1 to the original line's result into a separate register and adjusting the other register names accordingly in the file. We now have a suite of LLVM files and executables to run for each program line error. We then use the user-set individual line error probability, ERR_PROB, to determine the total probability that an error occurs during runtime according to the equation below, where N is the number of possible lines:

$$total\_error\_prob = 1 - (1 - ERR\_PROB)^N$$

The sdc_analyzer.py script then runs NUM_RUNS iterations of the executable, and each iteration will have a total_error_prob probability of executing one of the error-injected files, and a (1 - total_error_prob) probability of executing the correct synthesized executable. If an error-injected file is chosen, then the error-injected file corresponding to an error in a particular line is then randomly chosen according to an even distribution. The error is noted as a true error for later evaluation and analysis, compared to the actual detected errors during execution. Running on a real server will not yield a true error count. Each executable will run until either it is completed successfully, returning an error code of 0, or until the SDC detection code detects an SDC, returning a non-zero error code. The error code is used to pinpoint the exact line that the SDC occurred. With the detected distribution of errors per line, the true error distribution per line, and other metrics such as overhead, we calculate several metrics for evaluation, detailed in the Results section.

6 Results

We evaluate our tool by running the packaged LLVM compiler pass on 6 test C programs compiled to LLVM using clang. The error injection framework acts as an additional LLVM compiler pass that generates faulty LLVM code at all arithmetic instructions. The execution framework then executes the modified programs repeatedly, each time with a certain probability of executing each error-injected version or the non-injected version.

As shown using the simple pow2 example in Figure 7, the sdc_analyzer.py script generates two temporary modified LLVM files and executables, each corresponding to an error at the specified lines 29 and 38. The total_error_prob was calculated from the equation in the previous section to be 0.001999 from the individual ERR_PROB of 0.001 and N=2. A total of 222 errors were injected, with line distribution of 113 errors for line 38 and 109 errors for line 29. The number of SDCs found by our SDC detection system was also 222 errors, with the same distribution of errors in the pinpointed lines from the returned error codes, as the detector did not miss any SDCs in this simple example. Figure 8 displays a

```
Valid Instructions:
op1: %10
op2: 1
  %11 = shl i32 %10, 1 -> %dest0 = (bvshl %10 (int 1)) : i32
Running Racket script...
(bvshl %10 (int 1)) i32 ['bvshl'] %10 (int 1)
alt_rkt_code: (bvadd %10 %10)
type: i32
dest_reg: <rkt2llvm.DestReg object at 0x7f1afd7af2c0>
Alternate LLVM:   %dest_1 = add i32 %10, %10
Appending "  %dest_1 = add i32 %10, %10" before "  %11 = shl i32 %10, 1"
Appending "  call void @sdc_check_i32(i32 %11, i32 %dest_1, i32 1)" after "  %11 = shl i32 %10,
 1"
Instruction Time: 3.117724657058716 s

op1: %13
op2: 1
  %14 = add nsw i32 %13, 1 -> %dest1 = (bvadd %13 (int 1)) : i32
Running Racket script...
(bvadd %13 (int 1)) i32 ['bvadd'] %13 (int 1)
alt_rkt_code: (bvashr (bvsub %13 (bv #xffffffff 32)) (bv #x00000000 32))
type: i32
dest_reg: <rkt2llvm.DestReg object at 0x7f1afd7af2c0>
Alternate LLVM:   %dest_2 = sub i32 %13, 4294967295
  %dest_3 = ashr i32 %dest_2, 0
Appending "  %dest_2 = sub i32 %13, 4294967295
  %dest_3 = ashr i32 %dest_2, 0" before "  %14 = add nsw i32 %13, 1"
Appending "  call void @sdc_check_i32(i32 %14, i32 %dest_3, i32 2)" after "  %14 = add nsw i32
%13, 1"
Instruction Time: 3.060927152633667 s

{'  %11 = shl i32 %10, 1': '  %dest_1 = add i32 %10, %10', '  %14 = add nsw i32 %13, 1': '  %de
st_2 = sub i32 %13, 4294967295\n  %dest_3 = ashr i32 %dest_2, 0'}
Clang Conversion Successful: temp/pow2_mod.ll -> temp/pow2_mod_exe
Result: 16
Execution Successful: 0
Total Elapsed Time: 6.644347906112671 s
```

Figure 5. Terminal Output from llvm2racket.py for pow2.

graph for the distribution of detected errors. A more complex program like priority_queue would yield a more interesting distribution, as shown in Figure 9.

The program size, number of alternative instruction sequences generated, generation time, execution overhead, and error detection rate for each of the 6 test programs are shown in Table 1. Alternatives generated counts the number of alternatives found through program synthesis. Instructions for which no alternatives are found and simply duplicated in the program are not included. Generation time indicates the total time spent on program synthesis, running locally on laptops. Execution overhead is measured by the ratio of execution time running the modified (but not error-injected) versions of the programs to the original versions 10,000 times.

As shown in Table 1, the generation time is at most around one minute for a program with around 700 lines of code and 21 alternatives generated. The execution overhead is at most 3% in the worst case for all programs tested.

The error detection rate refers to the number of detected corruptions divided by the number of injected errors. It is not 100% because some errors are injected in some control paths of the program that is never executed in the current iteration.

7 Discussion and Limitations

While the instruction insertion tool is presented as a complete LLVM package, both the tool itself and the evaluation methodology have potential for future improvement.

The implementation of our tool inserts alternatives only for arithmetic instructions. Other important units, including memory and conditional instructions, are not checked. The program synthesis module can be extended to model memory and control path operations. For example, a load-word can be translated into two load-half-word instructions. Stores can be translated similarly and then checked with additional round-trip consistency comparison, as proposed in ITHICA [9].

As mentioned in the evaluation section, another limitation is that the total number of introduced errors included errors introduced in control paths that were never executed during the current iteration of the program. Therefore, while an error was injected into the system, there was no way for the SDC analyzer to detect the error, as the error was

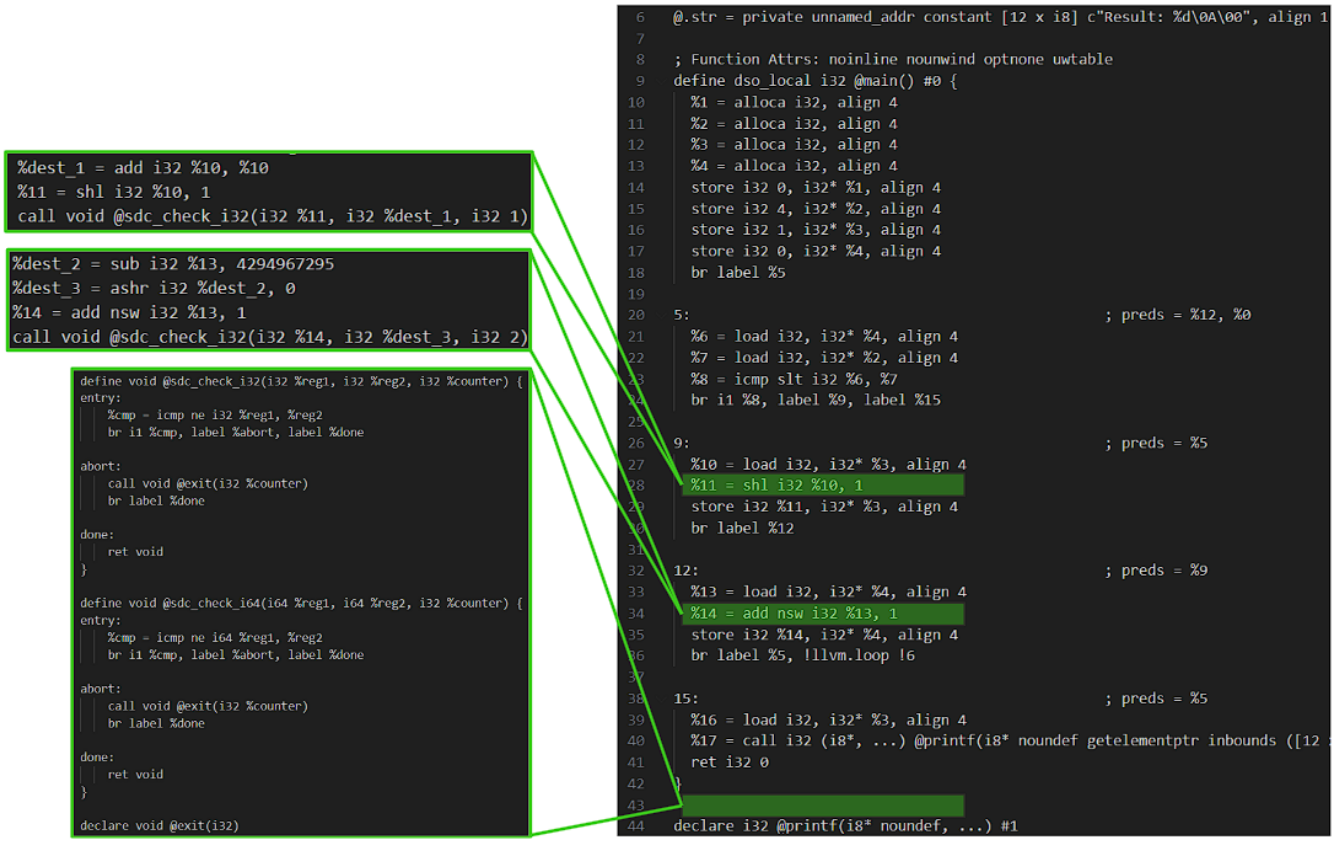

Figure 6. Insertion of LLVM Modifications (left) into Clang-generated LLVM (right).

Table 1. Evaluation result of the SDC detection tool as a compiler pass, running on 6 C programs, whose names indicate their functionality.

Test Program	#lines in original LLVM	#lines in modified LLVM	Alternatives generated	Generation Time (s)	Error Detection Rate (%)	Execution overhead (%)
backtrack.c	277	340	6	8.42	100	-0.93
basic_malloc.c	498	569	11	31.19	47	-0.92
bfs.c	724	825	21	60.53	84	0.93
pow2.c	62	110	2	2.74	100	2.73
prime.c	132	180	1	15.21	100	0
priority_queue.c	608	681	12	40.47	55	1.75

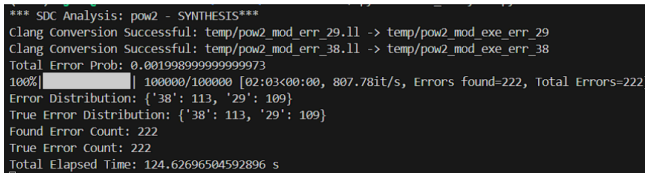

Figure 7. Terminal Output from sdc_analyzer.py for pow2.

never encountered in runtime. This appeared in the results as though the SDC analyzer "missed" the error, but any analyzer would not be able to detect an error not encountered. For more accurate results, we should only count the errors that were encountered during runtime rather than all the errors introduced. To resolve this problem, one can compute the metadata through test coverage programs and avoid injecting errors on uncovered lines. A better approximation is to use LLFI [12] as the error injection framework. We did not

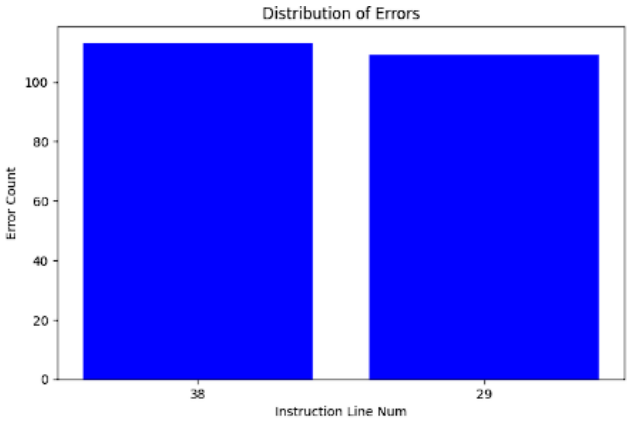

Figure 8. Graph Output from sdc_analyzer.py for pow2.

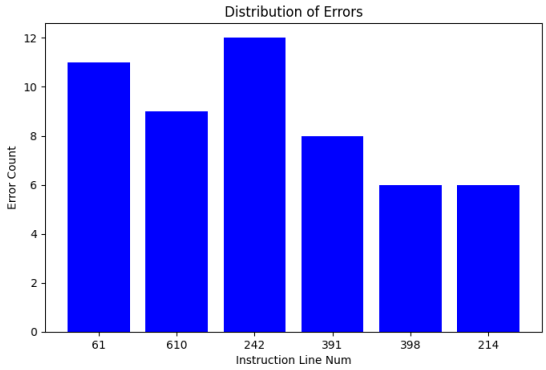

Figure 9. Graph Output from sdc_analyzer.py for priority_queue.

use LLFI because the repository is old and many packages are deprecated.

The experiments in this report were running on a Ubuntu/MacOS environment with the errors injected via another LLVM compiler pass. However, the errors artificially inserted at the LLVM level may not mirror the actual hardware defects. It is more accurate to use the real (defected and non-defected) servers and observe if corruptions are detected accurately.

The test programs used in the evaluation part are simple C programs that are not comparable in computation intensity to programs running on hyperscaler's server fleets. Complex tests, such as cpu-check [6], can be added as new test suites that target SDCs in the real world.

Acknowledgments

To Caroline Trippel and Rachel Cleaveland for their help throughout the class CS357S Formal Methods for Computer Systems, Winter 2025, at Stanford University.

A Artifacts

- Algorithm: Program Synthesis via Rosette. Software augmentation via LLVM compiler pass and Clang.
- Program: llvm2racket.py and sdc_analyzer.py
- Run-time environment: Linux x64, MacOS
- Hardware: multi-core x86-64 microprocessor, or MacOS ARM processor
- Output: Generated LLVM files, executable files, and graphs displaying analysis results
- Experiment: Each experiment is a run of sdc_analyzer.py for a C test program. Multiple C test programs can be evaluated
- How much time is needed to complete experiments (approximately)?: Around one hour is needed for generation of a program with around 700 lines of code. Program execution time is dependent on the original program, with an added overhead of at most 3% (experimentally).
- Publicly available?: Yes, at https://github.com/weixinyuxx/CS357S-Final.

A.1 Installation and Execution

1. Install Racket
2. Install Rosette
3. Install any other required dependencies
4. Clone Github repository at https://github.com/weixinyuxx/CS357S-Final
5. Change "file_name" variable in llvm2racket.py and sdc_analyzer.py to the name of the C test program name
6. Change "ERR_PROB" variable in sdc_analyzer.py to desired individual line error probability
7. Change "NUM_RUNS" variable in sdc_analyzer.py to desired number of runtime iterations
8. If testing on real server systems, change "INTRODUCE_ERRORS" variable in sdc_analyzer.py to False. If testing with artificially injected errors, change "INTRODUCE_ERRORS" variable in sdc_analyzer.py to True.
9. Run the two commands below to generate alternative LLVM code and run the analysis program:

Listing 1. Terminal Commands for Program Execution

```
% Run this line to generate alternative LLVM code
    with program synthesis
python3 llvm2racket.py

% Run this line to run and analyze the SDC
    detection
python3 sdc_analyzer.py
```

References

[1] Saranyu Chattopadhyay, Keerthikumara Devarajegowda, Bihan Zhao, Florian Lonsing, Brandon A D'Agostino, Ioanna Vavelidou, Vijay D Bhatt, Sebastian Prebeck, Wolfgang Ecker, Caroline Trippel, et al. 2023. G-QED: Generalized QED pre-silicon verification beyond non-interfering hardware accelerators. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[2] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data corruptions in the wild. arXiv:2203.08989 [cs.AR] https://arxiv.org/abs/2203.08989

[3] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245* (2021).

[4] Caleb Donovick, Jackson Melchert, Ross Daly, Lenny Truong, Priyanka Raina, Pat Hanrahan, and Clark Barrett. 2024. Peak: A single source of truth for hardware design and verification. *ACM Transactions on Embedded Computing Systems* (2024).

[5] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1109/SC.2012.49

[6] Google. n.d.. CPU-check torture test. https://github.com/google/cpu-check. Accessed: 2025-03-17.

[7] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 1–12. doi:10.1109/DSN.2012.6263960

[8] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. 2021. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 9–16.

[9] Caroline Trippel Ioanna Vavelidou (and maybe others). 2025. ITHICA: Intra-Thread Instruction Checking Approach for Defect-Induced Silent Data Corruptions. In *N/A*.

[10] Madeline Burbage Theo Gregersen Rachel McAmis Freddy Gabbay Baris Kasikci Jiacheng Ma, Majd Ganaiem. 2024. Proactive Runtime Detection of Aging-Related Silent Data Corruptions: A Bottom-Up Approach. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[11] Kevin Laeufer, Brandon Fajardo, Abhik Ahuja, Vighnesh Iyer, Borivoje Nikolić, and Koushik Sen. 2024. Rtl-repair: Fast symbolic repair of hardware design code. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 867–881.

[12] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. 2015. Llfi: An intermediate code-level fault injection tool for hardware faults. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 11–16.

[13] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. 2021. Silifuzz: Fuzzing cpus by proxy. *arXiv preprint arXiv:2110.11519* (2021).

Received 18 March 2025