FastFlip: Compositional SDC Resiliency Analysis

Abstract

To *efficiently* harden programs susceptible to Silent Data Corruptions (SDCs), developers need to invoke error injection analyses to find particularly vulnerable instructions and then selectively protect them using appropriate compiler-level SDC detection mechanisms. However, these error injection analyses are both expensive and monolithic: they must be run from scratch after even small changes to the code, such as optimizations or bug fixes. This high recurring cost keeps such software-directed resiliency analyses out of standard software engineering practices such as regression testing.

We present FastFlip, the first approach tailored to incorporate resiliency analysis seamlessly within the iterative software development workflow. FastFlip combines empirical error injection and symbolic SDC propagation analyses to enable fast and compositional error injection analysis of evolving programs. When developers modify a program, FastFlip often has to re-analyze only the modified program sections. We analyze five benchmarks plus two modified versions of each benchmark using FastFlip. FastFlip's compositional nature speeds up the analysis of the incrementally modified versions by 3.2× (geomean) and up to 17.2×. FastFlip selects a set of instructions to protect against SDCs that minimizes the runtime protection cost while protecting against a developer-specified target fraction of all SDC-causing errors.

1 Introduction

Processors are becoming increasingly susceptible to transient errors [7, 62]. The presence of Silent Data Corruptions (SDCs) in program outputs caused by such hardware-level errors during execution is difficult to detect. Software-level SDC detection techniques, such as instruction replication (e.g., [18, 44, 52]) are particularly attractive for protecting against SDCs, as they can be used on existing hardware. However, replicating all instructions leads to unacceptably high run-time overhead. To reduce run-time overhead to sustainable amounts, compilers can only *selectively* duplicate those instructions where errors are most likely to cause SDCs (e.g., [21, 31, 58]).

We can find vulnerable instructions using *instruction-level error injection analysis*, which injects errors one at a time into the dynamic instructions of a program during its execution, and records the effect on the output. For targeted protection, these analyses must provide *per-instruction* information on how errors in that instruction affect the output (e.g., [26, 61]), as opposed to just using sampling to provide overall statistical estimates of the program's vulnerability (e.g., [49, 59]). Such instruction-level resiliency analyses are time-consuming, requiring thousands of core-hours even for small programs.

This high analysis cost impedes the use of precise resiliency analyses in the iterative software development cycle, in which programmers regularly fix bugs, add features, and optimize their code. Each modification is frequently integrated into the code base, automatically compiled and tested to ensure the absence of bugs [63]. However, all previously proposed error injection analyses (e.g., [12, 13, 19, 29, 30, 36–38, 50, 55, 59–61]) must be re-executed on the full modified program after any – even minimal – code change, rendering them impractical.

We instead advocate for a *compositional* and *incremental* approach that partially reuses the results of error injection analyses from an old program version to reduce the analysis cost as the program evolves. We can divide the program into sections (e.g., function calls, code blocks, or loop nests), inject errors in each section separately, and combine the per-section results to get the program's SDC vulnerability results. When developers modify the program, we re-analyze only the parts of the code impacted by the change and reuse the results for the sections unaffected by the modification. This is the first step toward creating a software engineering discipline for hardware errors and resilience alongside the current, well-trodden, software engineering discipline for software bugs – ensuring hardening and functional correctness are both preserved.

Designing such an approach is challenging! The approach must propagate SDCs occurring within the output of one section through downstream sections to determine the SDCs in the final output. Similarly, errors in one section can corrupt memory locations that will be used only by subsequent sections, thus causing unexpected side effects without generating SDCs in the output of the current section. Finally, it should be general and support various existing resiliency analyses.

Our work. We present FastFlip, the first systematic approach for compositional and incremental error injection analysis of programs. FastFlip's theoretical foundation describes the conditions in which combining existing instruction-level error injection analyses and symbolic error propagation analyses is possible, and its practical framework specifies how to compute the impact of injected errors on a program's outputs. This allows FastFlip to leverage current and future advances in both analyses to efficiently find instructions vulnerable to SDCs.

When FastFlip first analyzes a program, FastFlip 1) *performs an error injection analysis* of each program section to find errors that cause SDCs, 2) *uses an SDC propagation analysis* to determine how SDCs propagate from one section to another to affect the final output, 3) *records the analysis results* for reuse on future program versions, and 4) *uses the analysis results to select a set of static instructions to protect* that minimizes runtime protection cost for a given target protection against SDC-causing errors. FastFlip also correctly accounts for side effects that only occur due to errors, and can adaptively adjust its results to meet SDC protection targets. When developers modify a program, FastFlip can reuse large portions of its analysis results: FastFlip only needs to rerun the expensive error injection analysis on the modified program sections and those downstream sections which receive a different input due to modified program semantics. By reusing FastFlip analysis results of other sections, FastFlip can save significant time.

For our evaluation, we instantiate FastFlip with 1) the Approxilyzer [61] per-instruction error injection analysis on top of an architectural simulator [60] and 2) the Chisel [43] SDC propagation analysis. We compare FastFlip against a baseline Approxilyzer-only error injection analysis that treats the entire program as a single section. For comparison, we use two key metrics used by previous work [25, 44, 52] on SDC protection via selective instruction duplication: 1) the *value* of protection (i.e., the coverage – the fraction of SDC-causing errors detected), and 2) the dynamic *cost* of protection (e.g., its runtime overhead). We analyzed each benchmark both before and after making two modifications. FastFlip provides a 3.2× speedup (geomean) over Approxilyzer for analyzing the modified programs, with minimal loss in protection value or increase in cost.

Contributions. This paper makes several contributions:

- *FastFlip:* We present FastFlip, the first approach for fast, compositional SDC error injection analysis of programs. FastFlip uses error injection and SDC propagation analyses to separately analyze program sections and then combine the analysis results. FastFlip then selects a set of instructions to protect to detect a target fraction of SDC-causing errors while minimizing the runtime cost of protection.
- *Instantiation:* We realize FastFlip using the Approxilyzer error injection analysis and the Chisel SDC propagation analysis. This combination allows FastFlip to analyze the effect of SDC-causing bitflips in architectural registers in the dynamic instructions of a program.
- *Evaluation:* We analyze five benchmarks with FastFlip, plus two modifications per benchmark (i.e., 15 versions total). FastFlip can analyze the modified benchmarks on average 3.2× faster and up to 17.2× faster than the Approxilyzer-only approach. For all benchmark versions, FastFlip successfully protects against the target fraction of SDC-causing errors for a similar cost as the Approxilyzer-only approach.

2 Background

2.1 Error injection analyses

Error injection analyses first find potential error sites at various points in an error-free execution of the program. These *error sites* can be bits in various registers, caches, etc. The analysis injects errors at each site one at a time and then executes the rest of the program, to record the effect of the error on the final output. Such analyses operate at different levels of abstraction, including hardware [19],, assembly [61], and IRt [12]. An error can have various effects on the program output:

- The error is *masked* – the program output is unaffected.
- The error causes a *crash* or other unrecoverable error causing the program to terminate unexpectedly.
- The error greatly extends the program runtime (e.g., by creating a long loop), causing a *timeout.*
- The error causes a *detectable* output change (e.g., by producing an incorrectly formatted output).
- The error changes the program output in an *undetectable* manner – known as a *Silent Data Corruption* (SDC).

The analysis result maps each error site to the outcome of an error at that site. Crashes, timeouts, and detectable output changes can be handled through relatively lightweight mechanisms such as checkpoints. SDC outcomes are more insidious and require more expensive methods such as task or instruction duplication for detection. However, many applications can tolerate small errors in their outputs (e.g., media/signal processing [43] and data science [24]). For such applications, it may not be necessary to protect instructions where errors mostly cause acceptably small SDCs (SDC-Good) and few unacceptably large SDCs (SDC-Bad). Thus, similar to Approxilyzer [61] we further classify SDCs as SDC-Good or SDC-Bad based on a developer defined, application-specific threshold ε. For applications that do not tolerate any SDC, ε is 0.

Analyses such as Approxilyzer aim to provide information on the outcome of errors at *all* error sites of a particular class within a program's execution on a specific input (e.g., [26, 61]). These instruction-level analyses are slower than approaches that do random sampling of error sites [5, 48], but can instead precisely identify vulnerable instructions that can then be protected/hardened at the compiler level [2, 18, 21, 28, 44, 52, 66].

2.2 SDC propagation analyses

SDC propagation analyses determine how SDCs within a program's input, or those introduced during execution, are propagated and amplified by the program up to the output. An SDC bound $\Delta(o) \leq f(\Delta(i))$ states that the SDC $\Delta(o)$ in the output o of a code section is at most a function f of the SDC $\Delta(i)$ in the input i. Many SDC propagation analyses, such as Chisel [43], conservatively and soundly analyze different control flow paths caused by SDCs and its impact on the output.

Sensitivity analysis [11] is a component of SDC propagation analyses that determines how sections of code amplify SDCs present within their inputs. In particular, *local* sensitivity analysis focuses on determining the effect of perturbations around a single input value. This analysis varies an input x_0 to a program section s by various amounts φ up to $\varphi_{\max}$. The analysis then executes s to calculate the output perturbation $|s(x_0+\varphi)-s(x_0)|$ and calculates the SDC amplification factor K, which is the Lipschitz constant [14] for s at x_0:

$$K = \max_{\varphi \leq \varphi_{\max}} \frac{|s(x_0+\varphi)-s(x_0)|}{\varphi} \tag{1}$$

K can be approximated by sampling a set of φ values [64] or a static analysis (e.g., [14, 17, 33]) can compute its upper bound.

3 Example

Lower-Upper decomposition (LUD) is a key matrix operation used in many applications. The blocked LUD algorithm consists of an outer loop with four sections that process various subsets of matrix blocks. We demonstrate FastFlip on the blocked LUD benchmark from the Splash-3 suite [53] for an example 16×16 input matrix with an 8×8 block size. In each iteration i of the loop, the loop body executes four sections in sequence that we call s_{i1},...,s_{i4}. Each section s reads the entire input matrix, but updates only the corresponding 8x8 block.

Hardware errors may occasionally occur in computations using this operation. While memory can be protected using ECC, but data currently being processed by the CPU is more vulnerable. If a bitflip causes an SDC, the corruption may not be detected and the user will receive a wrong answer. Here, we use a common error model is *single-event upset* (also used in [61]), which assumes that a only one bitflip occurs during the computation at a random bit in an architectural register within a random dynamic instruction in the program execution as the error site.

3.1 FastFlip Analysis

A developer can use an error injection analysis like Approxilyzer [61] (more details in Section 5.1) to systematically simulate errors to determine if the bitflips causes a SDC of output. While it gives a detailed map of vulnerable instructions, Approxilyzer requires over 600 core-hours for LU, and must be rerun from scratch after each modification to the program.

Fast Flip's Per-section analysis. Here, we describe how FastFlip calculates the SDC introduction and propagation characteristics (i.e., an *SDC specification*) of the 1st code section in the 2nd iteration of the LUD computation (we will call it $s21$) given its input data I_{21}. FastFlip repeats the following process for each section s of the full program T:

• FastFlip uses Approxilyzer on $s21$ in isolation to determine the effect of bitflips in each instruction in $s21$ aon its output O_{s21}. Some bitflips lead to SDCs in O_{s21}. We denote as φ_{s21} the magnitude of SDC introduced into O_{s21} due to one such bitflip.

• In addition, $s21$ can also amplify SDCs already present within its input I_{s21} due to a bitflip in previous computation. FastFlip uses a local sensitivity analysis to calculate the amplification level, e.g., if the magnitude of SDC present in I_{s21} is $\Delta(I_{s21})$, the resulting SDC in O_{s21} will be at most $f_{s21}(\Delta(I_{s21})) \leq 3.2\Delta(I_{s21})$.

• FastFlip combines these formulas to create a symbolic *SDC specification* for the section $s21$. Under the single bitflip error model, the *total* magnitude of SDC in O_{s21} ($\Delta(O_{s21})$) is upper-bounded by the sum of the propagated SDC ($f_{s21}(\Delta(I_{s21}))$) and the SDC potentially introduced by a bitflip in $s21$ (φ_{s21}):

$$\Delta(O_{s21}) \leq \varphi_{s21} + f_{s21}(\Delta(I_{s21})) \quad \textit{where} \quad f_{s21}(\Delta(I_{s21})) = 3.2\Delta(I_{s21})$$

Calculating an end-to-end SDC specification. FastFlip next provides these SDC specifications for all sections to Chisel [43], an SDC propagation analysis, plus a specification of data flow between sections (more details in Section 5.1).

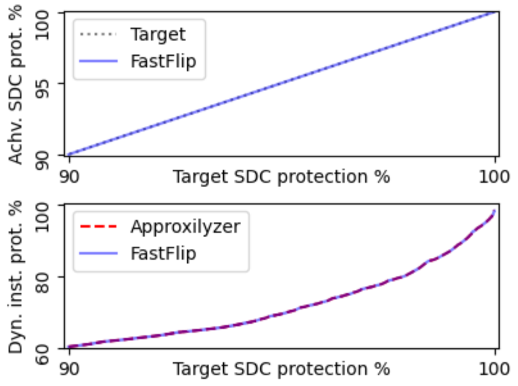

Figure 1. Protection value (top) and cost (bottom). There is the strong match of the FastFlip analysis to the developer's target value/cost within each plot (under 0.1% total difference).

Chisel uses this information to propagate potential SDCs caused by bitflips up to the final output and calculates the program's SDC specification. In this case, for two iterations of LU with four sections, Chisel calculates the following expression:

$$\Delta(O_{fin}) \leq 4174.8\varphi_{s11} + 434.3\varphi_{s12} + 28.8\varphi_{s13} + 3.2\varphi_{s14} + \varphi_{s21} + \varphi_{s22} + \varphi_{s23} + \varphi_{s24}. \tag{2}$$

Here φ_{sxy} is a symbolic variable representing the SDC potentially introduced into section y in iteration x. The numerical coefficient next to each φ_{sxy} represents the Chisel-calculated *total* amplification of φ_{sxy} by sections downstream of the error injection point (the coefficients depend on the program's input matrix data). FastFlip uses Equation 2 to propagate different SDCs from each section to the final output.

Selecting instructions to protect. FastFlip adapts the value and cost model from [25] to select a set of instructions to protect. FastFlip associates each static instruction pc with

- the *value* $v(pc)$ of protecting it, i.e., the number of SDC-causing bitflips at pc in the execution of program T, and
- the cost $c(pc)$ of said protection, i.e., the number of dynamic instances of pc in the execution of program T.

The value and cost of protecting a set of instructions is the sum of the value and cost of protecting each instruction in the set. This creates a trade-off space of total protection value and cost corresponding to each possible subset of instructions (see Figure 1; bottom plot). Given a target total SDC protection value, FastFlip aims to select a subset of instructions that minimize the total protection cost. This is a 0−1 knapsack optimization problem, which FastFlip solves via dynamic programming.

3.2 FastFlip Results

We compare FastFlip with Approxilyzer. We assume that a developer wants to protect against at least 90% of SDC-causing bitflips (getting coverage of 100% is often not achievable nor practical if the goal is selective protection). We compare how FastFlip's achieved result matches the target.

Value. The top plot in Figure 1 shows the value of protecting FastFlip's selection of instructions against SDCs. X-Axis shows the target value. Y-Axis shows the achieved value. The solid blue line shows FastFlip's achieved value, which overlaps the dotted black line showing the target value. FastFlip successfully achieves the target protection value for the entire target range.

Cost. The bottom plot in Figure 1 compares the cost of protecting FastFlip's and the Approxilyzer baseline's selection of instructions against SDCs. The X-Axis shows the target value, while the Y-Axis shows the protection cost. The red dashed line and solid blue line show the cost using Approxilyzer and FastFlip's results, respectively. The two lines overlap, and the excess of cost of FastFlip over Approxilyzer is below 0.1%.

Modifications. We next perform both analyses on two modified versions of this program. The *small* modification uses a specialized version of section 4 of the program which reduces the number of bounds checks when the matrix size is a multiple of the block size (as is the case for our input). The *large* modification replaces the first section with a lookup table. Unlike the baseline, which must inject errors in the full execution of the modified program, FastFlip only needs to inject errors in the modified program sections, saving considerable time. FastFlip's maximum deviation from the target value is 0.1% for these modified programs, and the excess of cost of FastFlip over Approxilyzer stays below 0.3%.

Analysis time. FastFlip requires 694 core-hours to analyze the original version of the program, compared to 602 core-hours for Approxilyzer. This is because Approxilyzer can *prune* error injections by forming equivalence classes of bitflips (i.e., bitflips that lead to the same outcome) across multiple sections. However, FastFlip saves significant time when later analyzing the modified versions of the program:

- For the version with a *small* modification (a few lines of code; see Section 5.5), FastFlip requires 80 core-hours, compared to Approxilyzer's 625 core-hours (7.8× faster).
- For the version with a *large* modification (replaces an entire section with a lookup table) FastFlip requires 94 core-hours, compared to Approxilyzer's 441 core-hours (4.7× faster).

This shows FastFlip's advantage in analyzing programs as they gradually evolve, saving time with each modification.

4 The FastFlip approach

Figure 2 visualizes the FastFlip approach. First, FastFlip performs two sub-analyses on each program section s in the full program execution T 1) FastFlip uses *an error injection analysis*[1] to determine the effect of each injection in s and stores the outcome, and 2) FastFlip uses*a local sensitivity analysis* to obtain an SDC propagation specification for s, and converts it into a total SDC specification for s. Second, FastFlip runs an SDC propagation analysis over T to obtain end-to-end SDC propagation specifications (we describe the properties of supported sub-analyses in Section 4.8). Third, FastFlip calculates concrete end-to-end SDC magnitudes to find the probability of an SDC-Bad outcome associated with each static instruction. Finally, FastFlip selects a set of instructions to protect with SDC detection mechanisms that minimizes the cost of protection, while also ensuring that the total value of the protection against SDCs is above a user-defined threshold.

4.1 Preliminaries

Definitions. We use the following symbols:

- T: dynamic trace of full program execution.
- s: section of the full program execution (usually a function call or execution of a code block or loop nest); $s \subset T$.
- J: set of all error injection sites in T.
- J_s: set of all error injection sites in s; $J_s \subseteq J$.
- $O_s(j)$: effect of an injection j on the outputs of s calculated by the error injection analysis.
- $i_{s,0},...,i_{s,m}$ and $o_{s,0},...,o_{s,n}$: inputs & outputs of s.
- $i_{T,0},...,i_{T,m}$ and $o_{T,0},...,o_{T,n}$: inputs & outputs of T.
- $f_{s,k}, f_{T,\lambda}, f_{T,\lambda,s}$: specifications of how the program sections propagate SDCs, calculated by the local sensitivity analysis, the SDC propagation analysis, and FastFlip respectively.
- $\varphi_{s,k}, \varphi_{*,*}, \varphi_{s,*}, \varphi_{\bar{s},*}$: symbolic variables (or sets thereof) for SDCs introduced into section outputs by errors.
- $p(j)$: probability that the error occurs at error site $j \in J$.
- $PC(j)$: maps $j \in J$ to the corresponding static instruction identifier pc. $PC(J)$ denotes the set of all static instructions of interest for error injection.
- ε_λ: maximum acceptable SDC for output $o_{T,\lambda}$ of T.
- $v(pc)$: the value of protecting static instruction at pc.
- $c(pc)$: the cost of protecting static instruction at pc.
- pc_{prot}: static instructions selected for SDC protection.

Analysis inputs. FastFlip accepts the full program T, its partition into sections s, a specification of how data flows between sections, the probabilities $p(j)$, SDC limits ε_λ, and protection cost function $c(\cdot)$ as inputs.

Sections are developer-identified parts of the program that perform specific tasks, like function calls, code blocks, or loop nests. Developers can obtain the dataflow specification using standard compiler analysis passes. Expert developers can also input this data manually (as we do).

Assumptions. As in previous works [26, 61], FastFlip assumes that 1) exactly one error occurs during the execution of the full program and 2) the program's input is SDC-free.

4.2 Error injection analysis of program sections

FastFlip runs an error injection analysis on each program section $s \in T$ to determine the effect of errors on the outputs of s. If an injected error j causes a detectable outcome (crash, timeout, misformatted output, etc.), then the outcome $O_s(j) = \textit{detected}$. Otherwise, the outcome $O_s(j) = (r_0, r_1, ..., r_n)$, where r_k is the magnitude of SDC (depending on the application and analysis, this can be absolute error, relative error, PSNR, etc.) caused by the injection j in output $o_{s,k}$ of s. If the injection is masked for an output $o_{s,k}$, then $r_k = 0$.

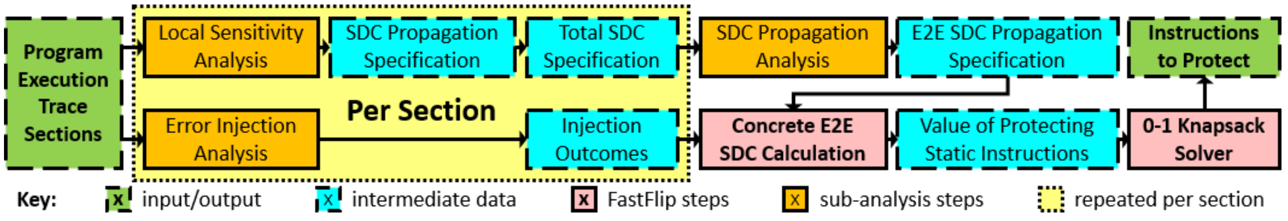

Figure 2. The FastFlip approach

4.3 SDC propagation analysis of program sections

FastFlip performs a local sensitivity analysis on each program section $s \in T$ to calculate how it amplifies SDCs present within its input. The local sensitivity analysis produces an SDC propagation specification for s of the general form:

$$\bigwedge_{k=0}^{n} \Delta(o_{s,k}) \leq f_{s,k}(\Delta(i_{s,0}),\ldots,\Delta(i_{s,m}))$$

For each output $o_{s,k}$ of s, the specification provides the SDC bound $\Delta(o_{s,k})$ calculated as a function $f_{s,k}$ of the SDC bounds of the inputs of s. FastFlip adds symbolic variables $\varphi_{s,k}$ to represent the magnitude of SDC introduced to $o_{s,k}$ during the execution of s as a result of an error within s. Under the single error model, if the input to s already contains SDC, then the error occurred in a previous program section, hence s cannot introduce additional SDC. Thus, we write the total SDC magnitude in the outputs of s as the sum of the SDC magnitude due to an error in s and the SDC propagated by s from its input to its output:

$$\bigwedge_{k} \Delta(o_{s,k}) \leq f_{s,k}(\Delta(i_{s,0}),\ldots,\Delta(i_{s,m}))+\varphi_{s,k} \tag{3}$$

4.4 End-to-end SDC propagation analysis

FastFlip runs an SDC propagation analysis on the full program T. FastFlip provides the analysis with the total SDC specifications from Equation 3 for each $s \in T$. The analysis also uses the developer-provided dataflow specification indicating how outputs of one section flow into the inputs of subsequent sections. The SDC propagation analysis uses this information to calculate the relationship between errors that occur anywhere in T to the SDC in the outputs of T. It creates an end-to-end SDC propagation specification of the form:

$$\bigwedge_{\lambda=0}^{n} \Delta(o_{T,\lambda}) \leq f_{T,\lambda}(\Delta(i_{T,0}),\ldots,\Delta(i_{T,m}),\varphi_{*,*})$$

where $\varphi_{*,*}$ is the list of all $\varphi_{s,k}$ variables from Equation 3 across all sections. Because the initial input to the full program is free of SDC, we can simplify $f_{T,\lambda}$ by removing all $\Delta(i_{T,*})$:

$$\bigwedge_{\lambda} \Delta(o_{T,\lambda}) \leq f_{T,\lambda}(\varphi_{*,*})$$

We next create specialized versions of $f_{T,\lambda}$ by noting that, under the single error model, the φ variables for only one section can be nonzero at a time: $f_{T,\lambda,s}(\varphi_{s,*}) = f_{T,\lambda}(\varphi_{s,*},\varphi_{\bar{s},*}=0)$. We rewrite the end-to-end SDC propagation specification as:

$$j \in J_s \Rightarrow \bigwedge_{\lambda} \Delta(o_{T,\lambda}) \leq f_{T,\lambda,s}(\varphi_{s,*}) \tag{4}$$

Equation 4 states that, if an error occurs in section s ($j \in J_s$), then the upper bound on the SDC in output $o_{T,\lambda}$ of T is given by $f_{T,\lambda,s}(\varphi_{s,*})$, a function of the SDC in the outputs of s.

4.5 Calculate value of protecting static instructions

FastFlip uses the injection outcomes (Section 4.2) and Equation 4 to answer the following question: *For a given static instruction identified by its program counter pc in the full execution T, what is the total probability of error injections in pc that result in SDC-Bad ($|SDC| > \varepsilon_\lambda$) for any output $o_{T,\lambda}$ of T?* This is the value $v(pc)$ of protecting pc.

Algorithm 1 shows how FastFlip calculates $v(pc)$. For each error injection in each section, FastFlip checks if the error results in a detectable outcome. If not, FastFlip calculates the RHS of Equation 4 to use as an upper bound on the magnitude of SDC in the outputs of T as a result of the error (i.e., the LHS of Equation 4). If the SDC in any output is SDC-Bad, FastFlip adds the probability of that error to the value of protecting pc. Lastly, FastFlip rescales the values so that the total value of protecting all static instructions is 1.

Algorithm 1 Find the value of protecting static instructions

```
Input   • $T$, $J_s$, $PC(j)$, $\varepsilon_\lambda$, $p(j)$: defined in Section 4.1;
        • $O_s(j)$: outcome of injection at $j \in J_s$;
        • $f_{T,\lambda,s}$: SDC propagation specifications from Equation 4
Returns $\forall pc.\ v(pc)$: value of protecting $pc$
1: $v \leftarrow \{\forall pc.\ pc \mapsto 0\}$
2: for $s$ in $T$ and $j$ in $J_s$ do
3:     $pc \leftarrow PC(j)$
4:     if $O_s(j) \neq detected$ then
5:         if $\exists \lambda.\ f_{T,\lambda,s}(O_s(j)) > \varepsilon_\lambda$ then
6:             $v(pc) \leftarrow v(pc) + p(j)$
7: $v_{total} = \Sigma_{pc} v(pc)$
8: $\forall pc.\ v(pc) \leftarrow v(pc)/v_{total}$
```

4.6 Finding an optimal set of instructions to protect

FastFlip uses the values $v(pc)$ calculated by Algorithm 1 for each pc and the corresponding protection costs $c(pc)$ as inputs to a 0−1 knapsack optimization problem. We model the value and cost of protecting a set of instructions as the sum of the value and cost of protecting each instruction in the set. Given a developer-defined target total protection value v_{trgt}, FastFlip solves the knapsack problem via the standard dynamic programming approach to select a set of static instructions pc_{prot} to protect that minimizes the total protection cost:

$$\boxed{\operatorname*{\mathbf{argmin}}_{pc_{prot} \subseteq PC(J)} \sum_{pc \in pc_{prot}} c(pc) \;\; \textbf{such that} \sum_{pc \in pc_{prot}} v(pc) \geq v_{trgt}}$$

We represent the set of all static instructions of interest $PC(J)$ as a binary vector, with one bit per static instruction. A bit in the vector is set if and only if the corresponding static instruction is in pc_{prot}. Under these conditions, the objective and constraints become linear functions of binary variables.

To explore the tradeoff space between value and cost, FastFlip selects the optimal pc_{prot} for a *range* of v_{trgt} values (e.g., $v_{trgt} \in \{0.9, 0.91, ..., 1.0\}$). This sequence corresponds to solving the value / cost multi-objective optimization problem using the ϵ-constraint method [42] (i.e., turning one of the objectives into a constraint as we did above) to obtain Pareto-optimal choices for pc_{prot}.

4.7 Composability

When developers modify a program section, FastFlip must rerun the error injection and local sensitivity analysis on the modified program section. If the modification changes the input to a downstream section by changing the modified section's semantics, FastFlip must also rerun these analyses on the affected downstream section. FastFlip can reuse the results of these sub-analyses for all other sections. FastFlip uses a data-flow analysis to identify such dependencies between inputs and outputs across sections.

Since the error injection analysis is the main contributor to analysis time, this approach significantly speeds up FastFlip compared to rerunning the error injection analysis on the full modified program, even when re-analyzing multiple sections.

4.8 Characteristics of compatible sub-analyses

Error injection analyses. The error injection analysis must separately report the outcome for errors in each error site in the program that the developer may wish to protect (e.g., [26, 61]), in contrast to just computing overall outcome statistics (e.g., [49]). This provides FastFlip with per-instruction error vulnerability information.

SDC propagation analyses. The SDC propagation analysis must support the SDC magnitude metric used by the error injection and sensitivity analyses. The analysis must also support the propagation of SDCs whose magnitude is represented by a symbolic variable. Examples of supported analyses are Chisel [43], DeepJ [33], and Daisy [16].

Error and cost models. FastFlip's formalism supports multiple error models. The error injection analysis may inject single or multi-bit errors into one or more error sites *within a single section*. The error sites can be individual instructions or coarser-grained program structures like statements. FastFlip also supports multiple cost models provided externally as a function $c(pc)$. This includes estimates of run time overhead for duplicating and comparing the results of single instructions (e.g., [52]) or instruction blocks, or the cost of specialized error detection for tasks or instruction blocks (e.g., [1, 2, 28]).

4.9 Factors that affect the precision of FastFlip

Inter-section masking. Inter-section masking occurs when an SDC present in one section is masked by a downstream section. FastFlip conservatively assumes that SDCs introduced in any section result in SDCs in the final outputs. The frequency of this masking is highly application-dependent.

Imprecision of sub-analyses. Imprecision in the error injection and SDC propagation sub-analyses used by FastFlip leads to imprecision in FastFlip. As FastFlip is a general approach that can use any sub-analysis that satisfies the requirements in Section 4.8, FastFlip's precision can be improved by using newer, more precise sub-analyses as they become available.

Side effects. A section may cause additional, unexpected side effects due to errors that do not occur in error-free executions. Consequently, the section outputs may be SDC-free, but the error may still cause SDCs in later sections. For example, the error may cause the section to overwrite data used by later sections due to bad memory address calculation, or the error may corrupt a register that will be used by later sections while popping it from the stack. FastFlip mitigates the first issue by considering variables in memory locations near the section output to also be section outputs, and the second issue by detecting the potential for such corruption and by considering the values in the popped registers to also be section outputs.

Untested error sites. A small number of error sites in the program may not be included in any program section. For example, if sections are executed multiple times within an outer loop, then the instructions which check the loop exit conditions may be excluded from all program sections. FastFlip conservatively assumes that, if an error occurs at such an untested error site, then it will *always* produce an SDC-Bad outcome. More rigorously, FastFlip creates a special section $s_\perp$ containing all such untested error sites j and assumes that $\forall j \in J_{s_\perp}, O_s(j) = (\infty, ..., \infty)$. This reduces precision, as not all untested errors sites result in an SDC-Bad outcome.

4.10 Adapting and compensating for loss of precision

A loss of precision due to the factors described in Section 4.9 leads to a loss of *utility*. That is, it can cause FastFlip to protect against a smaller number of SDC-causing errors than expected, or increase the cost of protecting FastFlip's selection of instructions beyond the minimum cost of protection. FastFlip adaptively adjusts the target value v_{trgt} used in Section 4.6 to compensate for this loss of utility (in our experiments, this adjustment is necessary for only one benchmark).

Measuring utility. FastFlip compares its utility to the utility obtained via a baseline monolithic error injection analysis. FastFlip uses two primary metrics for measuring utility. First, FastFlip treats the outcome labels of the baseline analysis as the ground truth and calculates the value of protecting its selection against SDC-Bad outcomes according to these alternate outcome labels. FastFlip refers to the protection value of its selection calculated in this manner as the *achieved value* v_{achv}. FastFlip then calculates the *loss of value* as $v_{loss} = v_{trgt} - v_{achv}$. v_{loss} measures the degree by which FastFlip undershoots v_{trgt}; a lower v_{loss} is better. Second, FastFlip calculates its excess cost over the baseline monolithic analysis. Specifically, if the costs associated with protecting the two selections of instructions against SDCs are c_{FF} (for FastFlip) and c_{Base} (for the baseline analysis) respectively, the excess cost is $c_{excess} = c_{FF} - c_{Base}$. c_{excess} measures the inefficiency of FastFlip's selection for protecting against SDC-Bad outcomes compared to the baseline analysis's selection; a lower c_{excess} is better.

When analyzing a program, FastFlip can simultaneously run the baseline error injection analysis for a minimal increase in analysis time. To do so, FastFlip checks the effect of each error in each section both on the section outputs and the final outputs. FastFlip efficiently calculates v_{loss} and c_{excess} using the outcome labels from FastFlip and the baseline analysis.

Adjusting the target value. FastFlip replaces the original target v_{trgt} with an adjusted target v'_{trgt}. Let the achieved value for this adjusted target be v'_{achv}. FastFlip minimizes v'_{trgt} such that $v'_{achv} \geq v_{trgt}$. If $v'_{trgt} > v_{trgt}$, then the cost of protecting FastFlip's selection increases, with larger adjustments leading to larger increases. If instead $v'_{trgt} < v_{trgt}$, the cost decreases.

Adjustment for modified program versions. If the number of modifications since the most recent target adjustment (m_{adj}) is below a developer-defined threshold (P_{adj}), FastFlip executes only its own time saving analysis and uses the existing adjusted target (v'_{trgt}) to choose the instructions to protect. As program modifications accumulate, the adjusted target may no longer provide the expected compensation for utility loss. Thus, once $m_{adj} \geq P_{adj}$, FastFlip recalculates v'_{trgt} by running a fresh analysis of the whole program while simultaneously running the monolithic analysis.

5 Methodology

5.1 Choice of sub-analyses

We instantiate FastFlip with the Approxilyzer [61] error injection analysis and the Chisel [43] SDC propagation analysis.

Approxilyzer is a bitflip error injection analysis that focuses on architectural CPU registers within each *dynamic* instruction in a program execution. Approxilyzer enumerates bitflip injection sites in the correct dynamic trace of the program for a particular input. It uses heuristics to form equivalence classes of bitflips that cause similar outcomes. Then, it injects a bitflip for a single pilot from each equivalence class into the correct execution of the program within the gem5 simulator, continues the now tainted program execution (with possibly incorrect control flow), and records the effect of the bitflip on the program output. It then applies the outcome of this pilot bitflip to all members of the equivalence class.

Chisel is an SDC propagation analysis that calculates the end-to-end SDC propagation function $f_{T,\lambda}$ as a *conservative affine function* of the symbolic SDC variables $\varphi_{*,*}$. Chisel conservatively assumes that each program section always amplifies input SDCs by the maximum amplification factor for that section for any input. Chisel supports diverging control flow paths by calculating the maximum possible SDC amplification over any path. Due to these assumptions, it generates conservative end-to-end SDC specifications. We added support for symbolic SDC variables to Chisel in order to calculate symbolic end-to-end SDC specifications for FastFlip.

5.2 Error model

We use the same error model as Approxilyzer [61], described below, to ensure a fair comparison. We inject one single-bit transient error per simulation in an architectural general purpose or SSE2 register. We target both source and destination registers in dynamic instructions within the region of interest, and skip instructions without register operands. We do not inject errors in special purpose, status, and control registers (e.g., %rsp, %rbp, %rflags) as we assume that they always need protection which can be provided by hardware. Similarly, we assume that caches are protected by hardware (e.g., ECC). As in previous works (e.g., [34, 59]), we assume that the probability $p(j)$ that the error will occur at any error site j is uniform.

5.3 SDC detection model

We assume that an instruction selected for protection is duplicated and then followed by an equality check of the results. The duplicated code and increased register pressure leads to runtime overhead. However, by rearranging instructions and checks, the overhead/cost for *extensive* instruction duplication across the program can be reduced to 29% on average [44] (*selective* duplication has even lower overhead, e.g. [31]).

Value and cost of detection. We adapt the value and cost model from [25]:

- The value of protecting a static instruction pc is proportional to the number of distinct errors injected in pc that produce an SDC-Bad outcome (using uniform $p(j)$).
- The cost $c(pc)$ of protecting pc is proportional to the number of dynamic instances of pc in the program trace.

5.4 Benchmarks

Table 1 presents our benchmarks, and we describe them next:

- *BScholes*: Black-Scholes analysis from PARSEC [6].
- *Campipe*: The raw image processing pipeline for the Nikon D7000 camera from [65].
- *FFT*: Fast Fourier Transform from Splash-3 [53].
- *LUD*: Blocked LU decomposition from Splash-3 [53].
- *SHA2*: The SHA-256 hash function from [45].

For FFT and LUD, the input size is the same as the minimized input size found by Minotaur [41], a technique for reducing injection time while retaining program counter coverage. For BScholes, we manually reduced the 21 option minimized input

Table 1. List of FastFlip benchmarks. The "Sections" Column shows static and dynamic instances of sections in the trace.

Benchmark	Input size	Sections	# Error Sites (|J|)
BScholes	2 options	4 (×2)	36.7K
Campipe	32×32	5 (×1)	72.7M
FFT	256×2	5 (×1)	9.23M
LUD	16×16	4 (×2)	1.75M
SHA2	32 bytes	3 (×1)	403K

found by Minotaur down to 2 options without reducing program counter coverage. For Campipe, we use the reference 32×32 input the implementation provided. For SHA2, we use a common cryptographic key size (256 bits).

5.5 Code modifications for benchmarks

To test the advantages offered by FastFlip for evolving programs, we analyze modified versions of each benchmark. Then, we compare the results of the baseline analysis (must re-analyze the whole program) to those of FastFlip (must only inject errors in modified sections). We experiment with two types of semantics-preserving modifications:

- **Small** modifications represent simple modifications that developers or compilers may make while optimizing and maintaining the program. Such modifications of up to 15 lines of code form a majority of open-source commits[3]. For Campipe and FFT, we store an expression used in multiple locations within the section into a variable to improve code readability. For LUD, we introduce a specialized version of a section that reduces the number of bounds checks if it detects that the matrix size is a multiple of the block size (as is the case for our input). For BScholes, we eliminate a redundant floating point operation in the cumulative normal distribution function. For SHA2, we similarly eliminate a redundant shift operation (without making the runtime input-dependent).
- **Large** modifications replace a program section with a lookup table. The table maps inputs of that section to corresponding outputs. If the modified section finds the current input in this table, it returns the corresponding output. Otherwise, it executes the original section code.

5.6 Baseline, comparison, and experimental setup

Software and hardware. FastFlip uses gem5-Approxilyzer version 22.1 [60] architecture simulator simulating an x86-64 CPU. We performed our experiments on AMD Epyc processors with 94 error injection experiment threads.

Region of interest. We focus on the computational portion of each benchmark and do not analyze I/O code.

SDC magnitude metric. We use the maximum element-wise absolute difference as the SDC metric. If $o_k[\ell]$ represents the ℓ^{th} element of an output o_k and the modified output due to an error is $\hat{o}_k$, then the SDC metric is $\max_\ell |o_k[\ell] - \hat{o}_k[\ell]|$.

SDC-Bad threshold. We first analyze all benchmarks assuming that any SDC is SDC-Bad ($\forall\lambda.\ \varepsilon_\lambda = 0$). Next, we relax this requirement in Section 6.4 by assuming SDC magnitudes up to 0.01 to be tolerable, i.e., SDC-Good ($\forall\lambda.\ \varepsilon_\lambda = 0.01$) for all benchmarks except SHA2 (whose applications require the output to be fully precise).

Sensitivity analysis parameters. As we consider the maximum tolerable SDC magnitude ε_λ to be 0.01 in Section 6.4, we use this as the maximum perturbation during the sensitivity analysis. To estimate the Lipschitz constant K (Equation 1), we perform 10^6 random perturbations up to ε_λ. For array inputs, we randomly perturb single, multiple, or all elements.

Comparison metrics. We compare the performance and utility of FastFlip to a baseline monolithic Approxilyzer-only approach. This baseline approach uses Approxilyzer to inject bitflips in the whole program at once and uses its results to select instructions to protect. For performance, we compare the analysis times of FastFlip and Approxilyzer.

For comparing utility, we compare the selections of instructions to protect made by the two approaches using the value and cost metrics. FastFlip only needs target adjustment for Campipe benchmark. For comparison, we choose three target values in the total value / cost trade-off space: $v_{trgt} \in \{0.90, 0.95, 0.99\}$, corresponding to protecting against 90%, 95%, and 99% of errors that cause unacceptably large SDCs.

Pruning error range. Approxilyzer's use of equivalence classes as described in Section 5.1 speeds up both FastFlip and the baseline analysis. However, the pilot is not a perfect predictor of the outcomes for the pruned injections (i.e., the rest of the equivalence class). Figure 5 in Approxilyzer [61] shows that, on average, 4% of pruned injections have an outcome that significantly differs from that of the pilot.

Therefore, we establish an error range around the achieved value of SDC protection to account for this discrepancy among the outcomes of injections in an equivalence class. This error range depends on the pilot prediction inaccuracy and the fraction of error sites with SDC-Bad outcomes that are protected. For FFT, LUD, and BScholes, we use the benchmark-specific pilot prediction inaccuracy from Figure 5 in Approxilyzer [61] (3%, 4%, and 10% respectively). For Campipe and SHA2, we consider the average inaccuracy from the same figure (4%). We give details of the error range calculation in the Appendix [4]. If v_{achv} is within or above this error range around v_{trgt}, then we consider FastFlip's result to be acceptable, even if $v_{achv} < v_{trgt}$.

Timeouts. FastFlip assumes that if the error causes the runtime of a program section to exceed 5× the nominal runtime, then the execution times out, which is a detected outcome. We use the same timeout rule for the Approxilyzer-only baseline.

6 Evaluation

6.1 Utility of FastFlip vs. Approxilyzer

Table 2 compares the utility of FastFlip and Approxilyzer for selective protection against SDCs, using the metrics described in Section 4.10. The pairs of columns show the utility comparison for the target protection values 0.90, 0.95, and 0.99 (90%, 95%, and 99% of SDC-causing errors) respectively. The

Table 2. Comparison of FastFlip and Approxilyzer utility when all SDCs are unacceptable (SDC-Bad). A ✓ indicates that the achieved value is within the value error range of FastFlip.

		$v_{trgt} = 0.90$		$v_{trgt} = 0.95$		$v_{trgt} = 0.99$	
Benchmark	**Modification**	**Value**	**Cost (diff)**	**Value**	**Cost (diff)**	**Value**	**Cost (diff)**
BScholes	None	0.901✓	0.635 (+0.000)	0.950✓	0.717 (+0.000)	0.990✓	0.827 (+0.000)
	Small	0.899✓	0.634 (+0.003)	0.950✓	0.713 (+0.000)	0.990✓	0.821 (+0.000)
	Large	0.898✓	0.669 (+0.000)	0.949✓	0.753 (+0.000)	0.991✓	0.849 (+0.000)
Campipe	None	0.915✓	0.611 (+0.038)	0.950✓	0.676 (+0.017)	0.991✓	0.807 (+0.024)
	Small	0.924✓	0.611 (+0.060)	0.954✓	0.678 (+0.030)	0.990✓	0.807 (+0.034)
	Large	0.912✓	0.760 (+0.068)	0.961✓	0.819 (+0.043)	0.993✓	0.899 (+0.015)
FFT	None	0.900✓	0.544 (+0.011)	0.950✓	0.629 (+0.002)	0.990✓	0.780 (+0.000)
	Small	0.904✓	0.542 (+0.010)	0.950✓	0.629 (+0.004)	0.990✓	0.781 (+0.002)
	Large	0.900✓	0.492 (+0.001)	0.950✓	0.586 (−0.000)	0.987✓	0.716 (−0.016)
LUD	None	0.900✓	0.603 (+0.000)	0.950✓	0.694 (+0.000)	0.990✓	0.873 (+0.000)
	Small	0.901✓	0.606 (+0.002)	0.951✓	0.698 (+0.002)	0.990✓	0.875 (+0.001)
	Large	0.902✓	0.560 (+0.002)	0.951✓	0.640 (+0.003)	0.990✓	0.826 (−0.001)
SHA2	None	0.900✓	0.666 (+0.001)	0.950✓	0.772 (+0.000)	0.990✓	0.908 (+0.001)
	Small	0.900✓	0.665 (+0.000)	0.949✓	0.771 (−0.001)	0.990✓	0.908 (+0.000)
	Large	0.883✓	0.476 (−0.007)	0.943✓	0.551 (−0.003)	0.985✓	0.655 (−0.007)

first column in each pair shows FastFlip's achieved protection value. The second column shows the cost of protecting FastFlip's selection, and compares this to the cost of protecting Approxilyzer's selection.

FastFlip successfully meets all target values for the unmodified (*None*) versions of each benchmark. Since FastFlip reuses the adjusted targets for the modified version it may not precisely meet the target for those modified versions. The maximum loss of value compared to the target is 0.017 (1.7%) for SHA2-Large. In all cases, the target value is within FastFlip's value error range caused by injection pruning. (We compare FastFlip with/without target adjustment in Section 6.3.)

For most benchmarks, the cost of protecting FastFlip's selection of instructions is at most 0.011 (1.1%) more than the cost of protecting Approxilyzer's selection. The exception is Campipe, for which FastFlip's cost is higher by as much as 0.068 (6.8%). Unlike the other benchmarks, FastFlip has to aggressively adjust the target values for Campipe in order to meet the original targets, to compensate for the loss of precision caused by inter-section masking. We observed that if we removed the last section of Campipe (the primary cause of inter-section masking), FastFlip's target adjustments became less aggressive. This suggests that more precise SDC propagation analyses that also calculate the probability of SDC masking may reduce the need for target adjustment.

The geomean cost of protecting FastFlip's selection is 0.601, 0.685, and 0.819 for the target protection values 0.90, 0.95, and 0.99, respectively. This shows that it is possible to protect against 90% bitflips that cause SDCs by protecting on average 60% of all dynamic instructions. Protecting against the remaining SDCs quickly leads to diminishing returns.

6.2 Performance of FastFlip vs. Approxilyzer

Table 3 compares the analysis time of FastFlip and Approxilyzer. Columns 1-2 show the benchmark name and version, respectively. Columns 3-4 show the analysis time for FastFlip and Approxilyzer *for that version of the benchmark*, respectively. Column 5 shows the speedup of FastFlip over Approxilyzer. We measure analysis time in *core-hours*. As the error injection analysis is highly parallelizable, the actual wall-clock time is much lower when using multiple CPU threads, but the wall-clock speedups show the same trend.

For FastFlip, error injection consumes over 99% of the analysis time. The sensitivity analysis requires less than five minutes of wall-clock time. The symbolic SDC propagation analysis and knapsack solver each require under one minute, even for programs/inputs much larger than our benchmarks.

To enable target adjustment, FastFlip simultaneously runs the Approxilyzer analysis as described in Section 4.10. We use the methodology from [41, Section 4.7] to confirm that the time required for this approach is at most 1% more than the greater of the analysis times of FastFlip and Approxilyzer for the unmodified versions of the benchmarks. As FastFlip reuses the adjusted targets for modified benchmarks, it does not need to use this approach when the benchmarks are modified.

The two approaches have similar analysis times for the unmodified (*None*) versions of all benchmarks except FFT. For FFT, Approxilyzer prunes a larger number of injections since it finds that an operation is repeated in different program sections. As FastFlip injects errors into each section independently, it cannot similarly prune injections across sections.

For the modified benchmarks, the speedup of FastFlip depends on the number of error sites that FastFlip must re-analyze

Table 3. Analysis execution time comparison

		Analysis time (core-hours)		
Benchmark	**Modif.**	**FastFlip**	**Approxilyzer**	**Speedup**
BScholes	None	69 hrs	65 hrs	0.9×
	Small	42 hrs	62 hrs	1.5×
	Large	3 hrs	24 hrs	8.4×
Campipe	None	2459 hrs	2631 hrs	1.1×
	Small	158 hrs	2720 hrs	17.2×
	Large	45 hrs	494 hrs	11.0×
FFT	None	980 hrs	520 hrs	0.5×
	Small	300 hrs	509 hrs	1.7×
	Large	93 hrs	513 hrs	5.5×
LUD	None	694 hrs	602 hrs	0.9×
	Small	80 hrs	625 hrs	7.8×
	Large	94 hrs	441 hrs	4.7×
SHA2	None	726 hrs	728 hrs	1.00×
	Small	718 hrs	726 hrs	1.01×
	Large	43 hrs	45 hrs	1.05×

compared to the full program. If the modified program sections represent a small fraction of the total error sites, then FastFlip provides large speedups. Critically, FastFlip is at least 1.7× faster when analyzing the modified versions of FFT. If the modified program sections represent a large fraction of the total error sites, then FastFlip provides smaller speedups. This leads to the negligible speedups for SHA2, where we modified the most expensive section of the program.

These results show that FastFlip can save significant time when analyzing evolving programs – here even a single reanalysis helped offset the original analysis overhead. For modern software systems that developers gradually modify multiple times, FastFlip provides ever increasing savings.

6.3 Effects of target precision adjustment

For all benchmarks except Campipe, the original and adjusted targets are virtually the same – the difference is below 0.4% of the original targets and all conclusion from Section 6.1, which presented results with adjustment, remain the same. For Campipe, target adjustment helps address the issue with the last stage described in Section 6.1.

Table 4 compares the utility of FastFlip and Approxilyzer for Campipe when FastFlip does not use target adjustment. The format is similar to that of Table 2, except that we omit the cost columns and focus on whether FastFlip still achieves the target value. Without target adjustment, FastFlip undershoots the targets by as much as 0.052 (5.2%) and the original target values do not always fall within FastFlip's achieved value error range. As Table 2 results for Campipe show, target adjustment resolves this issue and meets original protection target.

6.4 FastFlip support for acceptable SDCs

We next compare the utility of FastFlip and Approxilyzer when small SDCs (≤ 0.01) are considered acceptable (SDC-Good) and the analyses focus on protecting against errors that cause larger SDCs (SDC-Bad).

FastFlip successfully meets all target values for all benchmarks. The maximum loss of value compared to the target is 0.014 (1.4%) for FFT-Large. In all cases, the target value is within FastFlip's achieved value error range caused by injection pruning. For most benchmarks, the cost of protecting FastFlip's selection of instructions is at most 0.020 (2%) more than the cost of protecting Approxilyzer's selection. The exception is Campipe, for which FastFlip's cost is higher by as much as 0.057 (5.7%) due to aggressive target adjustment.

The geomean cost of protecting FastFlip's selection is 0.619, 0.720, and 0.849 for the target protection values 0.90, 0.95, and 0.99, respectively. FastFlip obtains the results for this scenario at the same time as the results in Table 2 for negligible additional analysis time (< 1 minute). We describe these results in more detail in the Appendix [4].

Table 4. Comparison of FastFlip and Approxilyzer utility for Campipe *before* target adjustment. A ✓ indicates that the achieved value is within the value error range of FastFlip, while a ✗ indicates the opposite. Table 2 shows the improved results after target adjustment.

		Value @ v_{trgt} =		
Benchmark	**Modif.**	**0.90**	**0.95**	**0.99**
Campipe	None	0.848✗	0.920✓	0.977✓
	Small	0.879✓	0.925✓	0.980✓
	Large	0.868✗	0.925✗	0.979✓

7 Related work

Error injection analyses. Error injection analyses operate at different levels of abstraction, including hardware, assembly, and IR [12, 13, 19, 29, 30, 36–38, 49, 50, 55, 59]. These analyses typically use *sampling* - they select a statistically significant number of error sites at random and only perform error injections at those sites. While this is sufficient for providing overall outcome statistics, a developer cannot use such results to determine which specific instructions or blocks of instructions are particularly vulnerable to SDCs in order to protect them. However, FastFlip can still use these analyses if they are modified to perform full instruction-level error injection like Approxilyzer [60, 61]. Minotaur [41] reduces the size of inputs (and therefore analysis time) required to test the reliability of programs when subjected to errors without compromising on the coverage of error sites. FastFlip further reduces analysis times for these minimized inputs as the program evolves.

Li et al. [37] show that error injection at higher levels of abstraction does not easily model the impact of all lower level hardware errors. Similarly, Papadimitriou and Gizopoulos [48] show that injecting errors in various SRAM hardware structures gives different results compared to injecting errors at higher levels of abstraction. AVGI [49] builds on [48] to show that hardware errors manifest in software in different ways, but result in similar distributions of final outcomes across applications. Santos et al. [54] similarly examine how faults injected at the RTL level affect common GPU instructions, and inject these effects into applications at the software level to provide overall outcome statistics and identify vulnerable hardware components. Unfortunately, such analysis techniques that aim to efficiently determine the effect of low-level faults via hybrid fault injection are too slow even for small program sizes when the outcomes are needed for *each* error site for fine-grained software-based SDC protection. If techniques such as AVGI become scalable for analyzing each error site in the future, we believe FastFlip would be able to use them.

Error injection-less reliability analyses. ePVF [20] is a dynamic analysis which finds locations where a bitflip will cause a crash, as opposed to an SDC, with ~ 90% accuracy. TRIDENT [35] uses empirical observations of error propagation in programs to predict the overall SDC probability of a program and the SDC probabilities of individual instructions. Several other works [9, 40, 56] use analytical modeling to detect SDCs in a program. While these analyses can be faster than error injection analyses, they are less accurate and may not be able to precisely estimate the magnitude of the output SDC due to an error. FastFlip's compositional nature makes error injection analysis more affordable by amortizing the cost of analyzing evolving programs over time.

SDC propagation analyses. SDC propagation analyses either propagate SDCs forward through programs [8, 10, 16, 23, 33], or propagate SDC bounds backwards through programs [22, 43]. While we instantiated FastFlip with Chisel [43] SDC propagation analysis, FastFlip can use other analyses that satisfy the conditions described in Section 4.8.

Mutlu et al. [46] predict the effect of bitflips injected into iterative applications on the final output by analyzing the effects of fault injections on a limited number of iterations. While this may give [46] an advantage over FastFlip for applications that iterate the same operation multiple times, unlike FastFlip, it cannot handle programs with multiple sections that perform distinct operations, such as our benchmarks.

Hardware-based selective protection. Researchers have examined the use of selective hardware hardening (e.g., via redundancy or ECC) for improving hardware reliability while limiting the use of additional chip area [15, 39, 51, 67]. These techniques find and replicate only those hardware components that, as a result of transient errors, produce unacceptable outcomes across the range of typical applications that users are expected to run on the hardware. FastFlip efficiently provides information which can be used to apply *additional*, software-based selective protection tailored to the needs of specific applications, as opposed to adding further hardware protections irrelevant to other applications.

Software-based selective SDC protection. Unlike crashes, timeouts, or clearly invalid data, SDCs are more difficult to detect by nature. SWIFT [52] uses instruction duplication to detect errors in computational instructions. To reduce overhead, it makes use of downtime in a program's instruction schedule. DRIFT [44] further reduces overhead by clustering together checks of multiple duplicated instructions to reduce basic block fragmentation. SWIFT and DRIFT *completely* eliminate the possibility of SDCs occurring due to single bitflip errors in the duplicated computational instructions. nZDC [18] provides comparable overhead to SWIFT while also protecting programs from 99.6% of SDCs caused by single bitflip errors during load, store, and control flow instructions.

Shoestring [21] finds and duplicates only particularly vulnerable instructions. Hari et al. [25] propose protecting blocks of instructions with single detectors placed at the end of loops or function calls. These two techniques use the results of error injection analyses to guide selective instruction duplication. Coarse-grained approaches place detectors at the task-level [2, 28, 66]. We consider such techniques to be *clients* of FastFlip. They provide FastFlip with information about the runtime overhead of protecting various instructions or instruction blocks. In return, FastFlip provides precise information on which instructions should be protected in order to minimize runtime overhead while protecting against a developer-defined fraction of SDC-causing errors. After these techniques protect FastFlip's selection of instructions, FastFlip can re-analyze the protected sections to confirm the decrease in SDC vulnerability. For FastFlip, we focused on efficiently handling program modifications in general. Testing code modifications specifically designed to reduce SDC vulnerability is an interesting topic for future work.

Incremental program analysis. Incremental techniques have a rich history in improving run-time of common program analyses that study control-flow equivalence and/or complex heap data structure properties, e.g., [27, 32, 47, 57]. In comparison, FastFlip incrementally analyzes the impact of hardware errors on the program execution, which are out of scope of off-the-shelf incremental techniques that operate on coarser-grained program properties.

8 Conclusion

We presented FastFlip, the first systematic approach that combines error injection and SDC propagation analyses to enable fast error injection analysis of evolving programs. When developers modify the program, FastFlip's compositional nature allows it to selectively re-analyze only the modified code sections, making it 3.2× faster on average (geomean) and up to 17.2× faster compared to a baseline non-compositional analysis that re-analyzes the whole program. Additionally, the value and cost of protecting FastFlip's selection of instructions is closely tracking that of the baseline analysis.

FastFlip can reduce the burden of repeated error injection analysis whenever developers fix program bugs, add optimizations, and add protections for vulnerable instructions. FastFlip therefore represents the first step toward including resiliency analysis and hardening as first-class citizens in the standard software development workflow, which continually compiles and tests software after each code modification.

References

[1] S. Achour and M. Rinard. 2015. Approximate Computation With Outlier Detection in Topaz. In *OOPSLA*.

[2] P. Agrawal. 1988. Fault tolerance in multiprocessor systems without dedicated redundancy. *IEEE Trans. Comput.* (1988).

[3] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. 2008. What's a Typical Commit? A Characterization of Open Source Software Repositories. In *ICPC*.

[4] Anonymous authors. 2024. Appendix to FastFlip: Compositional SDC Resiliency Analysis. https://doi.org/10.5281/zenodo.11122926.

[5] Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the propagation of transient errors in HPC applications. In *SC*.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*.

[7] Shekhar Borkar. 2005. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro* (2005).

[8] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. 2014. Uncertain<T>: A First-Order Type for Uncertain Data. In *ASPLOS*.

[9] Brett Boston, Zoe Gong, and Michael Carbin. 2018. Leto: verifying application-specific hardware fault tolerance with programmable execution models. In *OOPSLA*.

[10] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability Type Inference for Flexible Approximate Programming. In *OOPSLA*.

[11] Dan G. Cacuci and Mihaela Ionescu-Bujor. 2004. A Comparative Review of Sensitivity and Uncertainty Analysis of Large-Scale Systems - II: Statistical Methods. *Nuclear Science and Engineering* (2004).

[12] Jon Calhoun, Luke Olson, and Marc Snir. 2014. FlipIt: An LLVM Based Fault Injector for HPC. In *Euro-Par Workshops*.

[13] Chun-Kai Chang, Sangkug Lym, Nicholas Kelly, Michael B. Sullivan, and Mattan Erez. 2018. Hamartia: A Fast and Accurate Error Injection Framework. In *DSN Workshops*.

[14] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. 2011. Proving programs robust. In *ESEC/FSE*.

[15] Josie E. Rodriguez Condia, Paolo Rech, Fernando Fernandes dos Santos, Luigi Carrot, and Matteo Sonza Reorda. 2021. Protecting GPU's Microarchitectural Vulnerabilities via Effective Selective Hardening. In *IOLTS*.

[16] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs. In *TACAS*.

[17] Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *POPL*.

[18] Moslem Didehban and Aviral Shrivastava. 2016. NZDC: A Compiler Technique for near Zero Silent Data Corruption. In *DAC*.

[19] Waleed Dweik, Murali Annavaram, and Michel Dubois. 2014. Reliability-Aware Exceptions: Tolerating intermittent faults in microprocessor array structures. In *DATE*.

[20] Bo Fang, Qining Lu, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. 2016. ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis. In *DSN*.

[21] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *ASPLOS*.

[22] Vimuth Fernando, Keyur Joshi, and Sasa Misailovic. 2019. Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization. In *OOPSLA*.

[23] Vimuth Fernando, Keyur Joshi, and Sasa Misailovic. 2021. Diamont: Dynamic Monitoring of Uncertainty for Distributed Asynchronous Programs. In *RV*.

[24] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms*.

[25] Siva Hari, Sarita Adve, and Helia Naeimi. 2012. Low-cost Program-level Detectors for Reducing Silent Data Corruptions. In *DSN*.

[26] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *ASPLOS*.

[27] Kenneth Johnson, Radu Calinescu, and Shinji Kikuchi. 2013. An Incremental Verification Framework for Component-Based Software Systems *(CBSE '13)*.

[28] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2020. Aloe: Verifying Reliability of Approximate Programs in the Presence of Recovery Mechanisms. In *CGO*.

[29] Manolis Kaliorakis, Dimitris Gizopoulos, Ramon Canal, and Antonio Gonzalez. 2017. MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment. In *ISCA*.

[30] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, Nikos Foutris, and Dimitris Gizopoulos. 2015. Differential Fault Injection on Microarchitectural Simulators. In *IISWC*.

[31] Daya Shanker Khudia and Scott Mahlke. 2014. Harnessing soft computations for low-budget fault tolerance. In *MICRO*.

[32] Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. 2001. Incremental Verification by Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, Vol. 2031. 98–112.

[33] Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. 2022. A dual number abstraction for static analysis of Clarke Jacobians. *POPL* (2022).

[34] Guanpeng Li and Karthik Pattabiraman. 2018. Modeling Input-Dependent Error Propagation in Programs. In *DSN*.

[35] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling Soft-Error Propagation in Programs. In *DSN*.

[36] Jianli Li and Qingping Tan. 2013. SmartInjector: Exploiting intelligent fault injection for SDC rate analysis. In *DFT*.

[37] Man-Lap Li, Pradeep Ramachandran, Ulya R. Karpuzcu, Siva Kumar Sastry Hari, and Sarita V. Adve. 2009. Accurate microarchitecture-level fault modeling for studying hardware faults. In *HPCA*.

[38] Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. 2008. Online Estimation of Architectural Vulnerability Factor for Soft Errors. In *ISCA*.

[39] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech. 2019. Selective Hardening for Neural Networks in FPGAs. *IEEE Transactions on Nuclear Science* (2019).

[40] Qining Lu, Guanpeng Li, Karthik Pattabiraman, Meeta S. Gupta, and Jude A. Rivers. 2017. Configurable Detection of SDC-Causing Errors in Programs. *TECS* (2017).

[41] Abdulrahman Mahmoud, Radha Venkatagiri, Khalique Ahmed, Sasa Misailovic, Darko Marinov, Christopher W. Fletcher, and Sarita V. Adve. 2019. Minotaur: Adapting Software Testing Techniques for Hardware Errors. In *ASPLOS*.

[42] Kaisa Miettinen. 1998. *A Priori Methods*. Springer US, 115–129.

[43] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *OOPSLA*.

[44] Konstantina Mitropoulou, Vasileios Porpodas, and Marcelo Cintra. 2014. DRIFT: Decoupled CompileR-Based Instruction-Level Fault-Tolerance. In *LCPC*.

[45] Alain Mosnier. [n. d.]. SHA-2 algorithm implementations. https://github.com/amosnier/sha-2.

[46] Burcu O. Mutlu, Gokcen Kestor, Adrian Cristal, Osman Unsal, and Sriram Krishnamoorthy. 2019. Ground-Truth Prediction to Accelerate Soft-Error Impact Analysis for Iterative Methods. In *HiPC*.

[47] Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE*

Symposium on Logic in Computer Science, LICS 2018.

[48] George Papadimitriou and Dimitris Gizopoulos. 2021. Demystifying the System Vulnerability Stack: Transient Fault Effects across the Layers. In *ISCA*.

[49] George Papadimitriou and Dimitris Gizopoulos. 2023. AVGI: Microarchitecture-Driven, Fast and Accurate Vulnerability Assessment. In *HPCA*.

[50] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D. Antonopoulos, and Nikolaos Bellas. 2014. GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates. In *DSN*.

[51] Ilia Polian and John P. Hayes. 2011. Selective Hardening: Toward Cost-Effective Error Tolerance. *IEEE Design & Test of Computers* (2011).

[52] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. 2005. SWIFT: software implemented fault tolerance. In *CGO*.

[53] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *ISPASS*.

[54] Fernando F. dos Santos, Josie E. Rodriguez Condia, Luigi Carro, Matteo Sonza Reorda, and Paolo Rech. 2021. Revealing GPU's Vulnerabilities by Combining Register-Transfer and Software-Level Fault Injection. In *DSN*.

[55] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. 2015. FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In *11th European Dependable Computing Conference*.

[56] Vilas Sridharan and David R. Kaeli. 2009. Eliminating microarchitectural dependency from Architectural Vulnerability. In *HPCA*.

[57] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2021. Demanded abstract interpretation. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*.

[58] Anna Thomas and Karthik Pattabiraman. 2013. Error detector placement for soft computation. In *DSN*.

[59] Anna Thomas and Karthik Pattabiraman. 2013. LLFI: An Intermediate Code Level Fault Injector For Soft Computing Applications. In *QRS*.

[60] Radha Venkatagiri, Khalique Ahmed, Abdulrahman Mahmoud, Sasa Misailovic, Darko Marinov, Christopher W. Fletcher, and Sarita V. Adve. 2019. gem5-Approxilyzer: An Open-Source Tool for Application-Level Soft Error Analysis. In *DSN*.

[61] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V. Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *MICRO*.

[62] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. 2023. Understanding Silent Data Corruptions in a Large Production CPU Population. In *SOSP*.

[63] Wikipedia. [n. d.]. Continuous Integration. https://en.wikipedia.org/wiki/Continuous_integration.

[64] Graham R Wood and BP Zhang. 1996. Estimation of the Lipschitz constant of a function. *Journal of Global Optimization* 8 (1996), 91–103.

[65] Yuan Yao. [n. d.]. CAVA: Camera Vision Pipeline on gem5-Aladdin. https://github.com/yaoyuannnn/cava.

[66] A. Ziv and J. Bruck. 1997. Performance optimization of checkpointing schemes with task duplication. *IEEE Trans. Comput.* (1997).

[67] Christian G. Zoellin, Hans-Joachim Wunderlich, Ilia Polian, and Bernd Becker. 2008. Selective Hardening in Early Design Steps. In *ETS*.