Efficient Runtime Mercurial Core Detection with Core-Specific Test Case Synthesis

Jiacheng Ma
University of Michigan
jcma@umich.edu

Gefei Zuo
University of Michigan
gefeizuo@umich.edu

Andrew Quinn
University of California, Santa Cruz
aquinn1@ucsc.edu

Baris Kasikci
University of Michigan
barisk@umich.edu

I. Introduction

Cores do not always count correctly [11]. As semiconductor fabrication scales to smaller technology nodes, electrical defects are becoming more common due to the reduced gate size and the increased transistor density [4], [17], [18]. These defects result in *mercurial cores*—a set of CPU cores with malfunctioning instructions. Typically, a mercurial core does not crash immediately after failing a specific instruction, but produces a wrong computation result, which results in a variety of mis-behaviors including mis-computation, lock semantics violation, and data corruption [8]. Recently, major data center operators found that a very small subset of CPUs in their data centers have mercurial cores [8], [11].

Mercurial cores are risky, as they break the fundamental "fail-stop" assumption that software developers have been accustomed to for decades—a CPU either works correctly, or does not boot at all. Data center applications are built based on this assumption, and do not take CPU mis-behaviors like mis-computation into consideration. As a result, mercurial cores may lead to unpredictable failures.

Localizing mercurial cores is important; unfortunately, detecting these cores is hard due to two challenges. First, because electrical defects develop over time at different locations inside a circuit [13], any CPU core may become mercurial at anytime with any mis-behaviors and different mercurial cores may mis-behave in different ways. As a consequence, a test suite designed to detect mercurial cores must 1) be complex enough to cover enough logic inside the CPU, 2) be deployed in all CPUs in a data center, and 3) be executed periodically to detect newly developed defects. However, mercurial cores are rare; for most of the time, executing the test suite may only waste computing resources without finding any defects.

Second, the malfunctioned instruction of a mercurial core may only malfunction under certain conditions. For example, a defect caused by increased wire delay (e.g., due to electromigration) may be non-deterministic and only triggered under a certain voltage, frequency, and temperature [11], [14]. A core that is apparently not mercurial when the CPU is mostly idle may become mercurial when the CPU is running in production. As a result, a more meaningful mercurial core detection should ideally be conducted at runtime in production, e.g., before executing critical code sections.

Previous work on this field does not fully address these challenges. These works can be classified into two categories: *fuzzing-based techniques* and *scan-based techniques*.

Fuzzing-based techniques, such as Google's SiliFuzz [16], use a fuzzer to generate a comprehensive test suite and deploy it at scale in a data center. With SiliFuzz, Google managed to find "multiple" (undisclosed number) mercurial cores in their production clusters, demonstrating the effectiveness of fuzzing-based techniques. Unfortunately, fuzzing-based techniques rely on enormous tests to detect defects effectively (e.g., SiliFuzz generates around 500,000 tests). Thus, these tests are not suitable to be run frequently at runtime.

Scan-based techniques, such as ACE [7] and Intel In-Field Scan (IFS) [1], leverage a software-accessible scan-chain to detect electrical defects. With a scan-chain, developers can feed carefully-constructed test patterns into each flip-flop inside a circuit, execute the circuit for a few cycles, and read back the updated values from individual flip-flops; by comparing the read-back values against the desired ones, developers can accurately identify malfunctioned flip-flops and gates. Unfortunately, scan-based techniques are unlikely to cover complex defects that only occur under certain voltage, temperature, system noise, and workload [6], [15].

To address the aforementioned challenges, we propose Defecticon, a system for efficient mercurial core detection at runtime. Unlike previous work which uses the same test suite for all CPUs in a data center, Defecticon customizes test cases for individual cores based on their own characteristics. Besides, Defecticon provides an API for developers to embed these test cases into their applications, enabling mercurial core detection at runtime.

In the rest of this paper, we discuss the key insights (§II) and design details (§III) of Defecticon.

II. Key Insights

We design Defecticon based on two insights:

First, while different mercurial cores may have different mis-behaviors, each core tends to mis-behave in a consistent way. This allows us to tailor the test case for each core guided by the characteristics of their defects; by only deploying specialized test cases, the execution time of mercurial core detection can be significantly reduced, enabling frequent testing at runtime.

Second, while some defects may be non-deterministic, they are more likely to be triggered under a lower voltage [9], [10]. By performing scan-based tests with an intentionally reduced, abnormally low voltage, we can obtain a set of flip-flops and gates that are more likely to fail first; we can then use this information to synthesize test cases customized for each core.

III. Design of Defecticon

The workflow of Defecticon comprises of two phases: a *potential defect discovery* phase (§III-A)—which identifies flip-flops/gates that are more prone to becoming defective than others—and a *defect lifting* phase (§III-B)—which generates instruction sequences that induce software-visible errors if the discovered potential defects are really defective.

A. Potential Defect Discovery

In the potential defect discovery phase, Defecticon performs a series of scan-based tests under decreasing voltages. Like traditional scan-based testing, Defecticon's scan testing has three steps: *scan-in* (feeding carefully-constructed test patterns into each flip-flop), *testing* (executing the core), and *scan-out* (reading back values in each flip-flop).

Defecticon uses a normal voltage (i.e., a voltage within the CPU's normal operation range) for scan-in and scan-out, and uses an intentionally reduced, abnormal voltage for testing. Defecticon repeats the test for multiple rounds and reduces the testing voltage in each round until it finds a number of failing flip-flops and gates. Defecticon marks these failing flip-flops and gates as potential defects.

Notably, scan-chains are commonly used in CPUs for testing purpose; thus, the test patterns fed into the circuit and tools to determine the locations of failing flip-flops and gates are already available. Furthermore, as recent and future Intel processors provide software interfaces for voltage control [2], [12] and scan-based testing [1], it becomes feasible to conduct scan-based tests in field with a reduced voltage.

B. Defect Lifting

In the defect lifting phase, Defecticon performs RTL-level analysis on the CPU design that contains the potential defects discovered in §III-A. Defecticon targets generating an instruction sequence for the core that produces different output states (e.g., values in registers and cache) when a defect is triggered versus when it is not triggered.

For each potential defect, Defecticon instruments the circuit with a new data path from the potential defects to the output, simulating the scenario when the defect is triggered. Considering the following code snippet as an example:

```
always @(posedge clk) o <= (x & y) | z;
```

In the code snippet, `x`, `y`, and `z` are the inputs and `o` is the output. Assuming the `&`-operation of `x` and `y` is potentially defective, Defecticon instruments the code snippet with the following lines of code:

```
1 always @(posedge clk) o_def[0] <= 0 | z;
2 always @(posedge clk) o_def[1] <= 1 | z;
3 always @(posedge clk) o_def[2] <= ~(x & y) | z;
```

These lines of code simulate the three types of failures when the potential defect is actually triggered—the computation result of `x&y` sticks at 0, sticks at 1, or is flipped. The new outputs `o_def[0]`, `o_def[1]`, and `o_def[2]` are generated, representing the output value when the defect is triggered under different failure models.

For each generated output simulating the result of a defect (e.g., `o_def[0]`), Defecticon uses symbolic execution [5], [19] to search for an instruction sequence that leads to a difference between that generated output and the original one (e.g., `o`). A symbolic execution engine treats the input of a circuit as symbolic, mathematical value and executes the circuit mathematically. During symbolic execution, the value of each register/wire will be computed as a mathematical expression. When a constraint is applied to some values, the engine can generate a concrete example (i.e., with a specific value instead of an expression) of the input values that satisfy the constraint.

For the above code snippet, Defecticon applies the constraint $\texttt{o\_def[0]} \neq \texttt{o}$ for Line 1, $\texttt{o\_def[1]} \neq \texttt{o}$ for Line 2, and $\texttt{o\_def[2]} \neq \texttt{o}$ for Line 3. After symbolic execution, Defecticon will find out that the values of `z` must be 0 to satisfy the constraints for all lines; for Line 1, both `x` and `y` must be 1; for Line 2, at least one of `x` and `y` must be 0; for Line 3, the value of `x` and `y` does not matter. On a real CPU core, Defecticon uses a similar method to obtain a set of constraints on the input instruction sequences; it invokes the constraint solver to generate an example sequence of instructions satisfying these constraints.

However, modern implementations of CPU cores can be complex; as a consequence, the symbolic execution engine may not be able to find a desired instruction sequence due to state space explosion [3], a common problem in symbolic execution. Defecticon uses heuristics to shepherd the symbolic execution engine towards a desired state; for example, it may ask the symbolic execution engine to only explore a new state if it results in more registers being influenced by the defect at each cycle. Similarly, Defecticon may also use heuristics to bound the search space; for example, it may assume the instruction sequence contains a fixed number of instructions.

For a specific CPU core, we assume the number of potential defects is small (e.g., a single digit number), such that the generated instruction sequence is lightweight enough to be invoked before and after executing each critical section in an existing application. As a result, developers can have the confidence that the CPU core is working properly during the critical section. Defecticon provides the utilities and APIs to generate the proper assertions and invoke core-specific test cases at application runtime.

References

[1] https://www.kernel.org/doc/html/latest/x86/ifs.html.

[2] https://github.com/torvalds/linux/blob/143a6252e1b8ab424b4b293512a97cca7295c182/drivers/platform/x86/intel/turbo_max_3.c.

[3] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[4] M. T. Bohr and I. A. Young, "Cmos scaling trends and beyond," *IEEE Micro*, vol. 37, no. 6, pp. 20–29, 2017.

[5] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224, http://dl.acm.org/citation.cfm?id=1855741.1855756.

[6] H. H. Chen, "Beyond structural test, the rising need for system-level test," in *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, 2018, pp. 1–4.

[7] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects mechanisms, architectural support, and evaluation," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 97–108.

[8] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[9] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* Citeseer, 2003, pp. 7–7.

[10] H. Hao and E. J. McCluskey, "Very-low-voltage testing for weak cmos logic ics," in *Proceedings of IEEE International Test Conference-(ITC)*. IEEE, 1993, pp. 275–284.

[11] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 9–16.

[12] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "{V0LTpwn}: Attacking x86 processor integrity from software," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1445–1461.

[13] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 93–104. [Online]. Available: https://doi.org/10.1145/1555754.1555769

[14] P. G. Ryan, I. Aziz, W. B. Howell, T. K. Janczak, and D. J. Lu, "Process defect trends and strategic test gaps," in *2014 International Test Conference*, Oct. 2014, pp. 1–8.

[15] P. G. Ryan, I. Aziz, W. B. Howell, T. K. Janczak, and D. J. Lu, "Process defect trends and strategic test gaps," in *2014 International Test Conference*. IEEE, 2014, pp. 1–8.

[16] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "Silifuzz: Fuzzing cpus by proxy," *arXiv preprint arXiv:2110.11519*, 2021.

[17] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 389–398.

[18] A. J. Strojwas, K. Doong, and D. Ciplickas, "Yield and reliability challenges at 7nm and below," in *2019 Electron Devices Technology and Manufacturing Conference (EDTM)*. IEEE, 2019, pp. 179–181.

[19] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for validating the security of processor designs," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 815–827.