Probabilistic Bug-Masking Analysis for Post-Silicon Tests in Microprocessor Verification

Doowon Lee†, Tom Kolan‡, Arkadiy Morgenshtein‡, Vitali Sokhin‡,
Ronny Morad‡, Avi Ziv‡ and Valeria Bertacco†

†University of Michigan
{doowon, valeria}@umich.edu

‡IBM Research – Haifa
{tomk, arkadiym, vitali, morad, aziv}@il.ibm.com

ABSTRACT

Post-silicon validation has become essential in catching hard-to-detect, rarely-occurring bugs that have slipped through pre-silicon verification. Post-silicon validation flows, however, are challenged by limited signal observability, which impacts their ability of diagnosing and detecting bugs. Indeed, bug manifestations during the execution of constrained-random tests may be masked and be unobservable from the test's outputs. The ability to evaluate the bug-masking rate of a test provides great value in generating and/or selecting effective tests for high coverage regressions.

To this end, we propose an efficient, static bug-masking analysis solution, called BugMAPI. BugMAPI tracks the information flow in a test program, and it estimates the probability that bugs go undetected by the checking mechanisms in place in the post-silicon platform. To achieve this goal, we leverage static code analysis and a novel, lightweight, probability estimation algorithm. We evaluated BugMAPI on a range of industrial constrained-random tests and a range of bug injection models, and we found that it can estimate bug-masking rates with an accuracy of 77% in 3 orders-of-magnitude less time, compared to an ideal dynamic analysis solution.

CCS Concepts

•**Hardware → Bug detection, localization and diagnosis;** *Simulation and emulation;*

Keywords

Post-Silicon Validation, Random Tests, Bug Masking

1. INTRODUCTION

Advances in semiconductor technology enable us to integrate billions of transistors in a single chip. This ever-growing complexity poses a challenging problem for microprocessor design verification: indeed, reaching coverage closure within reasonable time is becoming extremely difficult [6]. High-end microprocessors often include complex features (*e.g.*, transactional memories, security enhancements and multiprocessor memory consistency) that are hard to verify in the early stages of verification. Therefore, post-silicon validation, that is, the validation effort carried out on the first silicon prototypes, aims at catching all the remaining bugs that were not detected in the pre-silicon stage [11]. One of the most difficult aspects in the deployment of post-silicon validation, however, is the extremely limited observability into the design's internals. This aspect makes bug detection and diagnosis challenges of their own. Even more importantly, bugs may manifest during a test's execution, but become masked and go unnoticed by the time the test completes. The outcome in these situations is that the post-silicon test simply wastes precious prototype's execution cycles.

DAC '16, June 05-09, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4236-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2897937.2898072

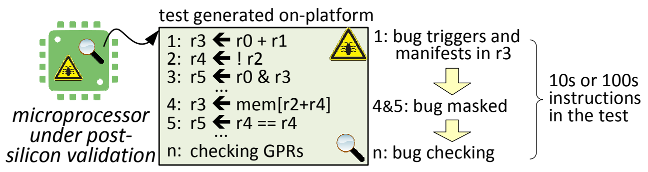

test generated on-platform
1: r3 ← r0 + r1
2: r4 ← ! r2
3: r5 ← r0 & r3
...
4: r3 ← mem[r2+r4]
5: r5 ← r4 == r4
...
n: checking GPRs
microprocessor under post-silicon validation
1: bug triggers and manifests in r3
4&5: bug masked
n: bug checking
10s or 100s instructions in the test

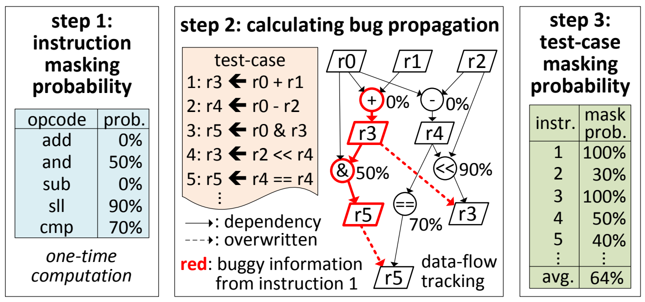
Figure 1: **Bug-masking problem in post-silicon random instruction tests**. In this example code, the results of instructions 1 and 3 are overwritten by instructions 4 and 5, so that any bug occurring or propagating through the former two instructions would not be detected at the end of the test.

The types of tests deployed in post-silicon validation vary widely, from application snippets, to compatibility tests, to constrained random tests. These latter ones are particularly valuable in trying to exercise corner-case scenarios, since a vast number of variants can be generated with little designer's effort [1,10,15]. One additional benefit that they bring to the validation effort is that they can often be generated directly in the microprocessor under verification (*i.e.*, on-platform), enabling an efficient use of the usually scarce number of platforms and prototypes available.

But even in light of this rich set of options for tests, bug masking still remains a significant problem in post-silicon validation. Bug masking occurs when, *due to post-silicon's limited observability, a bug becomes undetectable after some amount of computation has occurred past its manifestation.* Indeed, in post-silicon validation, it is often possible to only monitor architectural registers and memory, usually only at the end of the test execution, while in pre-silicon validation, more detailed information is available (*e.g.*, cycle-based micro-architectural state). An example of this problem is illustrated in Figure 1, where the first instruction triggers a silicon bug, leading to an incorrect value being written in register *r3*. However, the erroneous value in *r3* is subsequently overwritten, and thus undetectable by the time the test completes.

During the post-silicon validation process, identifying situations that may lead to bug-masking provides great benefits in improving the quality of the validation effort. For instance, as we demonstrate in Section 7, it allows to guide the application of small perturbations to tests (synthetic or from real-applications) so to minimize masking effects, attaining higher coverage from a regression suite. Other applications of a bug-masking evaluation include assessing the quality of a test: tests to be included in key regressions can be selected based on having a low bug-masking incidence. In addition, in test generation, test-templates can be selected and designed to limit the generation of sequences that are susceptible to masking.

In this paper, we propose BugMAPI, **Bug**-**M**asking **A**nalysis with **P**robabilistic **I**nformation-flow. BugMAPI is a static code analysis



opcode	prob.
add	0%
and	50%
sub	0%
sll	90%
cmp	70%

instr.	mask prob.
1	100%
2	30%
3	100%
4	50%
5	40%
⋮	⋮
avg.	64%

Figure 2: **BugMAPI overview**. BugMAPI applies static instruction-flow analysis to instruction test sequences by leveraging accurate, pre-computed masking probabilities for each instruction, and by deriving overall test-sequence masking rates using information-flow tracking.

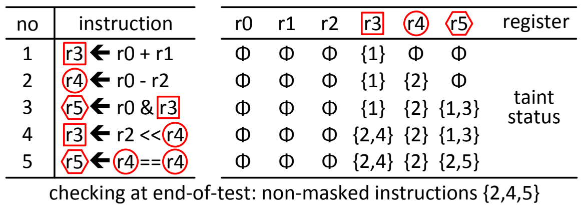
for instruction test sequences. It allows to estimate the probability that bugs go undetected by the checking mechanisms available on the validation platform. It analyzes information-flow paths (*i.e.*, use-def chains in data-flow analysis) existing in the test-case, tracking the exposure to information loss. BugMAPI is inspired by static information-flow tracking analyses, and it improves on them by (1) leveraging a process based on tailoring the bug-masking probability to each specific instruction and by (2) developing a heuristic to derive the bug-masking exposure of an instruction sequence from the individual instruction probabilities. In summary, we make the following contributions:

- We propose a novel, static bug-masking analysis solution that provides high accuracy at low computational cost by leveraging a heuristic masking-risk computation algorithm.
- We showcase an application of our analysis that greatly reduces the bug-masking incidence in constrained-random tests.

2. RELATED WORK

Test-case generation for microprocessor verification. Randomly generated test-cases are often used to trigger any hard-to-find, unexpected bugs in microprocessor validation [1,15]. For instance, [1] creates random test-cases by leveraging designer-made test-templates, each targeting a specific test scenario. The tests often select high-quality instructions (*i.e.*, those likely to discover bugs) among those specified in the templates by solving constraint satisfaction problems. Other approaches leverage formal models, often specified through architectural description languages [10]. Researchers have also proposed solutions that create tests with self-checking properties, so to overcome low observability in post-silicon validation without additional instrumentations (*e.g.*, scan chain, design-for-debug network) [7,9,16]. These self-checking approaches use either reversing [16], equivalent [7], or repeated operations [9]; however, they may not reflect realistic program sequences.

Information-flow analysis has been investigated extensively in the computer security area [3,12,14]. The information flow occurs when information is transferred among distinct program objects (*e.g.*, variables) [3]. To secure the program execution, access-control policies enforce confidentiality throughout the program [14]. In this context, taint analysis (*e.g.*, [12]) takes a dynamic approach that uses run-time information to detect confidentiality violations. Our bug-masking analysis adopts the principle of information-flow from the security area, but with a completely different goal: we aim at identifying perturbations due to bug occurrences that may be erased by the program's subsequent computation. Moreover, in our case, all instruction resources are potential sources of bugs, while in taint analysis only data from untrusted devices is tracked. Information-flow analysis has also been used in soft-error analysis [5], but without leveraging an instruction's unique characteristic to accurately



no	instruction	r0	r1	r2	r3	r4	r5	register
1	r3 ← r0 + r1	Φ	Φ	Φ	{1}	Φ	Φ	
2	r4 ← r0 - r2	Φ	Φ	Φ	{1}	{2}	Φ	
3	r5 ← r0 & r3	Φ	Φ	Φ	{1}	{2}	{1,3}	taint status
4	r3 ← r2 << r4	Φ	Φ	Φ	{2,4}	{2}	{1,3}	
5	r5 ← r4 == r4	Φ	Φ	Φ	{2,4}	{2}	{2,5}	

checking at end-of-test: non-masked instructions {2,4,5}

Figure 3: **Baseline information-flow analysis**. Our baseline analysis propagates information flow leveraging instructions' inputs and outputs. In the best case, checkers at the end of the test could reveal any bug that occurred at instructions 2, 4, or 5. Bugs manifesting at instructions 1 or 3 are masked.

estimate the propagation probability. Finally, [4] shares with us the deployment of information-flow analysis concepts to hardware verification by annotating RTL designs to compute observability coverage metrics.

3. BUGMAPI OVERVIEW

BugMAPI applies static code analysis to instruction test sequences. The results of the analysis report which instructions in the sequence are likely to mask potential bugs occurring during their execution. To this end, we first introduce a basic approach in Section 4; we then improve on it in Section 5 by taking into account the type of instruction in the analysis.

Figure 2 outlines BugMAPI. Our solution considers a test sequence and applies three steps to it: the first step assigns a bug-masking probability to each instruction in the test sequence, based on its functionality (Section 5.1). The second step tracks data dependencies between instructions and computes the probability that data affected by a bug propagates to the end of the test, thus remaining observable by end-of-test checkers. It also leverages an approximation technique to reduce the computational complexity of this step (Section 5.2). The last step derives the bug-masking probability of the overall test, from those computed for the individual instructions.

Our endeavor focuses on improving the accuracy of a typical static analysis, by leveraging an accurate masking-probability computation for each instruction. Note that, to be efficient, BugMAPI does not take into account concrete values (*e.g.*, operand values in arithmetic instructions) in determining if a bug is masked by an instruction. Overall, our static approach reduces computation requirements by a few orders of magnitude, compared to a simulation-based dynamic analysis. This benefit comes at the cost of lower accuracy, due to lack of dynamic information.

4. BASELINE ANALYSIS

We first developed a simple information-flow tracking technique that determines the observability of buggy values in registers and memory, which we refer to as our **baseline analysis**. This analysis is static, in contrast to popular taint-tracking solutions in the security domain [3,12]. Our choice is driven by the need of post-silicon validation environments to be efficient and keep a low-computation profile, since the execution of the test itself is extremely fast.

Figure 3 illustrates an example of the baseline analysis, applied to a 5-instruction sequence. For each instruction, we extract inputs and outputs as per the ISA specification. Then, for each resource in the test program (*e.g.*, registers), we proceed instruction-by-instruction and compute which instructions can propagate their taint status to it. In other words, instruction X propagates the taint status to register Y if a bug manifesting in instruction X could affect the value of register Y [3]. For instance, after the first instruction, *r3* has a taint list containing instruction 1, while all other registers have empty lists. Note that the third instruction uses the *r3* value computed in the first instruction and writes to register *r5*, so that the taint status

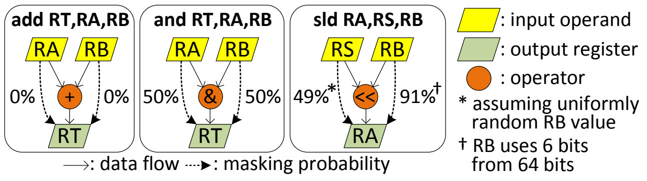


Figure 4: **Examples of input-output bug-masking probabilities**.

of *r5* becomes $\{1, 3\}$. At the end of the analysis (last line), the taint status lists indicate which instructions' results impact the status of each register at the end of the test program. Thus, by computing the union of all the taint lists, we find which instructions could expose bugs that are not masked by the program.

Note that inaccuracies are possible because this baseline analysis does not take into account the distinct operation of each instruction. For instance, with reference to Figure 3, at the end of the program, *r3* should reveal bugs occurring during the execution of the second instruction. However, if $r2 = 0$, any erroneous value stored in *r4* would be masked in the fourth instruction, clearing the taint in *r3*. Consequently, our baseline analysis provides conservative bug-masking results and false-negatives (*i.e.*, bugs predicted as detectable, but masked in reality) may indeed occur. In the next section, we try to overcome precisely this limitation.

5. PROBABILISTIC MASKING ANALYSIS

BugMAPI refines the baseline analysis from the previous section by taking into account the instruction type in computing bug-masking probabilities. As an example, Figure 4 shows this refinement for three instructions from the Power ISA [8] (*add*, *and* and *sld*). For the sake of this example, let's assume that bugs manifest as single-bit flips in a source register (Sections 6.1 and 6.5 discuss the actual bug-models used in our evaluation). The goal here is to compute the probability that a single-bit flip in input registers RA, RB or RS propagates to the output register value. In the case of *add*, the bit flip propagates in all cases, except for overflow, so masking probability is practically 0%. In the case of an *and* instruction (middle of Figure 4), there is a 50% chance that the bit flip is masked by a 0-value in the corresponding bit position of the other operand. For *sld* (shift left doubleword), we computed that the bugs in the shift amount RB are masked 91% of the execution times (because only the least significant bits are used by the instruction), while the source register RS is masked 49% of the times. Note that the baseline analysis is equivalent to assigning a 0% bug-masking probability to all the input-output flow paths, hence it is clear that the baseline analysis would lead to a much higher false-negative rate.

BugMAPI computes the bug-masking probabilities for each instruction of a processor's ISA using the process discussed below in Section 5.1. This computation is only carried out once per ISA. It then considers the test-case under analysis, applies the individual probabilities to each instruction and computes how they propagate to the end of the test program, similarly to our baseline analysis (details are in Section 5.2). Finally it can compute the masking probability of the entire program, for instance, as an average masking rate over the individual contributions of the instructions in the program. Note that the last step can be tailored to the specific validation goal. For example, when computing the average, the contribution of specific instructions can be weighted to tilt the criticality of masking to certain units of the processor (*e.g.*, floating point), or certain high-interest program constructs.

5.1 Masking through an Instruction

When computing the masking probability of an instruction, we consider each input and output (i/o) pair and compute the probability independently for each pair. Note that it is possible that the probabilities through multiple i/o pairs are correlated. For instance, the *addo* instruction in Power ISA sets the overflow flag when appropriate. Thus a single buggy input operand may affect both the target register and the overflow flag. However, to keep our computation manageable, we disregard these second-order effects.

We developed two approaches to compute masking probabilities of instructions. The first is an analytical approach, where we investigate the ISA specification (*e.g.*, [8]) to understand the specific functionality of each instruction, and compute the masking probability for each i/o pair mathematically. We used this model mostly for branch and load/store instructions.

The second is an experimental approach whereby we execute the instruction in an instruction-set simulator (*e.g.*, gem5 [2]) with 1,000 sets of random input values. For each i/o pair, we run each input set twice, one with the random input generated and the other with a buggy version of the input set, and then we compare the output to determine if the bug was masked. Based on the overall findings, we can derive the masking probability with high confidence. To attain efficiency in this process, we develop assembly programs that encapsulate the input generation, the repeated executions and the probability computation, all together. Developing those programs is the only part of this approach that requires engineering effort.

Both solutions have their own advantages. The experimental approach can be more accurate, and it can be easily adapted to a new bug model. The analytical one is beneficial when developing the assembly program is too time-consuming, and for instructions whose masking probabilities tend to be unaffected by the specifics of the bug model.

5.2 Masking over an Instruction Sequence

The goal of this step is to compute, for each instruction in a sequence, the likelihood that a bug manifesting at the instruction propagates to the end of the sequence, revealing an incorrect test outcome. Computing this likelihood accurately is computationally intensive: a simulation-based dynamic analysis would have to consider a vast number of executions to properly randomize all the input sources for all the instructions in the sequence. A static analysis could be very involved because of how a buggy instruction can have a large fanout, impacting many other registers and resources over the test execution, all of which would have to be tracked to determine if they eventually become masked.

Because of the high cost of an accurate analysis, we chose to design a high-quality approximation for it. The calculation makes three simplifying assumptions, which we discuss below and illustrate with an example in Figure 5. Note that for sake of simplicity, we perform the analysis using bug-propagation probabilities, which are the complement of bug-masking ones.

Assumption 1 — *Instructions are independent among each other.* As a result, we can calculate the bug-detection probability through multiple instructions, by simply calculating the product of each instruction's probability of propagating an erroneous value. As shown in Figure 5 (dark orange region), a bug in *r0* propagates to *r4* through instructions 1 and 2. Consequently, the probability that a bug manifests in *r4* through *r0* is the product of the probability of having a bug in *r0* before the program starts, times the two instructions' bug-propagation probability.

Assumption 2 — *Inputs of an instruction are independent among each other.* Because of this assumption, we derive the bug propagation likelihood to an output as the complement of the probability that both i/o pairs are masked. With reference to the example in Figure 5 (yellow area), the probability that a bug propagates from either *r2*

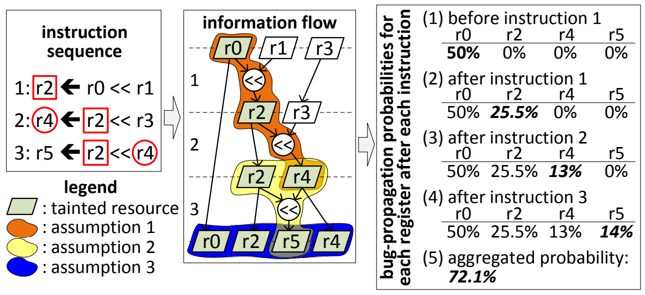


Figure 5: **Example of bug-propagation analysis over an instruction sequence**. Initially, only *r0* carries a buggy value with a 50% probability. Through the execution of instructions 1–3, the buggy value may propagate also to *r2*, *r4* and *r5*. We derive the propagation probabilities on the right table by leveraging our three simplifying assumptions.

or *r4* to *r5* is: $P((r2 \to r5) \cup (r4 \to r5)) = 1 - P((r2 \to r5)^c \cap (r4 \to r5)^c) = 1 - (1 - P(r2 \to r5)) \times (1 - P(r4 \to r5)) = 1 - [1 - P(r2) \times P(RA \to RT)] \times [1 - P(r4) \times P(RB \to RT)] = 1 - (1 - 0.255 \times 0.51) \times (1 - 0.13 \times 0.09) = 14\%$.

Assumption 3 — *Resources are independent among each other.* This assumption is useful in calculating the final bug-propagation probability from an instruction through an entire sequence. It allows to compute the overall bug-propagation probability from the probability that the bug had propagated to any of the monitored resources (*e.g.*, registers). With reference to the example, in the blue region of the figure, we indicate that the final resources available are *r0*, *r2*, *r4* and *r5*. *r0* is included because it is the only register that propagated the bug until before instruction 1. By the end of the code snippet, bugs in *r0* may be reflected in erroneous values in *r0* or any of the other registers in the blue area. Using the independence assumption, and a similar calculation as in the previous paragraph, the probability that the bug manifests at the end of the sequence is: $P(r0 \cup r2 \cup r4 \cup r5) = 1 - P(r0^c \cap r2^c \cap r4^c \cap r5^c) = 1 - (1 - P(r0)) \times (1 - P(r2)) \times (1 - P(r4)) \times (1 - P(r5)) = 1 - (1 - 0.5) \times (1 - 0.255) \times (1 - 0.13) \times (1 - 0.14) = 72.1\%$.

6. EXPERIMENTAL EVALUATION

In this section, we first discuss our bug models (Section 6.1) and computational cost (Section 6.2). We then present BugMAPI's characterization (Section 6.3) and accuracy (Section 6.4) by comparing against a dynamic analysis, and conclude with a brief discussion (Section 6.5).

BugMAPI is implemented in Python, and it is evaluated with test-cases generated for two ISAs: IBM Power and DEC Alpha. For IBM Power, we generated bug-masking probabilities for approximately 4,000 i/o pairs corresponding to all instructions for the POWER8 processor. Note that our implementation for Power does not accurately take into account memory address disambiguation, and it assumes that store operations have no overlapping addresses. For DEC Alpha, however, we did implement a simple disambiguation mechanism by dedicating a pool of registers only to load/store base addresses. We only included a subset of 118 instructions and 218 i/o pairs in our Alpha ISA evaluation.

We used Threadmill [1] to generate IBM Power tests, and a simple in-house test-generator for Alpha ISA tests. Note that BugMAPI is not yet capable of handling multi-threaded tests, analyzing only one thread at a time. We discuss multi-thread related issues in multi-threaded tests in Section 6.5.

6.1 Bug Models

The goal of our bug model is to capture many corner-case functional bugs, representative of those often detected in post-silicon

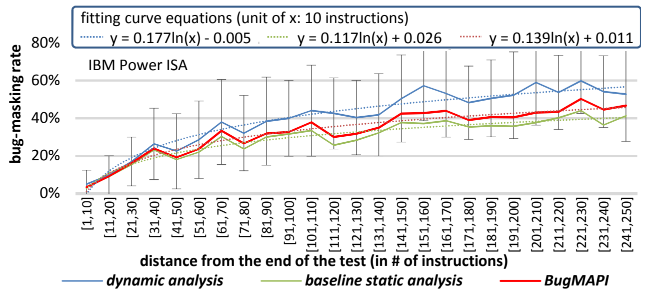


Figure 6: **Bug-masking rate for individual program instructions**, sorted by increasing distance from the end of the test program. Each data point is obtained by averaging over all instructions in the distance window and over 30 distinct randomly-generated tests.

validation. To this end, we inject bugs that *slightly alter the architectural state* (*i.e.*, registers and memory locations). Note that this model mimics micro-architectural bugs that ultimately modify the architectural state. For instance, a malfunctioning cache will be eventually revealed when its incorrect data is read by a load operation. However, purely micro-architectural bugs that lead to execution delays or bugs that are usually detected by system's hangs cannot be detected by BugMAPI and thus our models are not concerned with capturing them.

In our reference dynamic analysis, we inject bugs by modifying a register value, or a memory location, right after it has been updated. We considered five types of modifications that could be forced by the bug, each corresponding to a distinct bug model, as we discuss in Section 6.5. After a preliminary analysis, we settled for the *random-value* option because we believe that it resembles more closely what occurs in practice. With this model, a register value or a memory location affected by a bug is replaced by a random value. Note that when injecting bugs into memory locations, we are careful to generate a value matching the original bit-width. For instructions that trigger multiple updates, we only inject a bug in one output value at a time.

6.2 Performance Analysis

We evaluated BugMAPI's execution time on a dual Intel Xeon system. To analyze 100 instances of the longest Alpha test (the last bar of Figure 7), BugMAPI took approximately 11.3 seconds. The dynamic analysis for the same tests took 61 minutes of ISS simulations using 7 cores (one simulation per each bug injection, running one core per simulation). From these values, we estimate that BugMAPI provides a *3-orders-of-magnitude speedup* over a dynamic analysis. In addition, the probability computation described in Section 5.1 took approximately 2 minutes for the subset of instructions of the Alpha ISA and less than one hour for all the instructions of the POWER8 ISA. Note that this step is only required once per ISA, and is easily absorbed over the large number of tests evaluated.

6.3 Bug-Masking Characterization

In Figure 6, we plot the bug-masking rate of an instruction as a function of its distance from the end of the test program for IBM Power ISA. This evaluation was carried out to gather a sense of how much program length affects masking probabilities, and to place the accuracy of BugMAPI with respect to a dynamic analysis and a baseline static analysis. Each data point is obtained as the average of the bug-masking rate over all instructions in the windows indicated: for example, the first data point on the left averages the last 10 instructions of the test, while the rightmost one averages the instructions that are between 241 and 250 instructions before the end of the test. Moreover, each data point is obtained by averaging

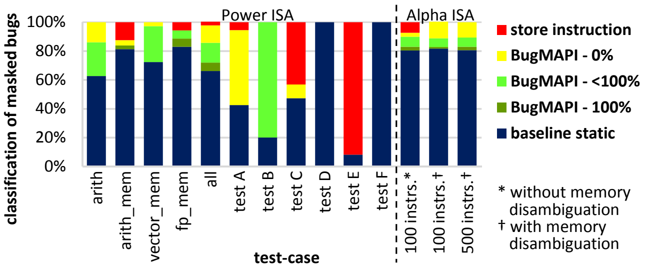


Figure 7: **Classification of masked bugs**. Masked bugs are classified based on the ability of our baseline analysis and BugMAPI to identify the masking.

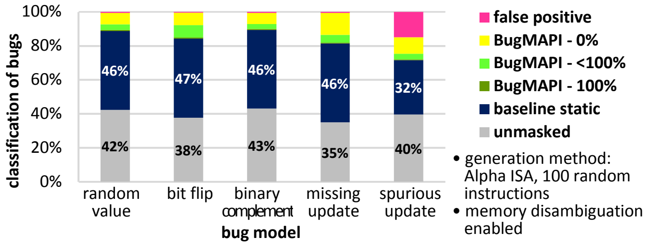


Figure 8: **BugMAPI's accuracy on various bug models**.

over 30 distinct test programs.

The diagram compares three solutions: a dynamic analysis (where bugs are detected by comparing against an equivalent execution with no bug injection), the static baseline analysis, and BugMAPI. As expected, the bug-masking rate is higher for instructions further away from the end of the test where the checking occurs. However, it is encouraging to notice the quantitative values of this trend: even after 200 instructions, the average bug has only a masking probability of 40–50%, based on BugMAPI's and the dynamic simulation's estimates. Note that BugMAPI's analysis is only 21% less accurate (by comparing fitting curve equations) than the dynamic one, while achieving a 3-orders-of-magnitude speedup.

6.4 Accuracy

To evaluate BugMAPI's accuracy, we deployed a perfectly accurate dynamic analysis for a number of test programs and then classified each bug that this analysis found to be masked, based on BugMAPI's analysis of that same test. The tests we considered are 11 industry-strength tests for Power ISA and 3 fully random tests for Alpha ISA. Five of the Power ISA tests were generated using templates that select instructions randomly from a given set: *arith* uses only arithmetic fixed-point instructions, while *arith_mem* selects from loads and stores as well as arithmetic ones. *vector_mem* and *fp_mem* choose instructions from vector and floating-point instructions, respectively. The remaining six tests target the memory subsystem including memory consistency and address translation. One test program has been generated for each template, varying in length from 26 instructions (test *D*) to 917 instructions (*all*). For the Alpha ISA, we report average results over 100 tests with the specified characteristics. For example, in the tests "*without memory disambiguation*," memory addresses are always treated as non-overlapping.

Our classification provides the following categories:

1. *baseline static*: Bug found masked by the baseline static analysis.
2. *BugMAPI – 100%*: Bug found masked by BugMAPI with 100% probability, but not included in the previous group.
3. *BugMAPI – <100%*: Bug found masked by BugMAPI with less than 100% probability.
4. *BugMAPI – 0%*: Bug found NOT masked by BugMAPI.
5. *store instruction*: Bug injected in store instructions or other instructions affecting a store instruction. This corresponds to the accuracy penalty due to lack of memory address disambiguation.

Figure 7 reports the findings of our analysis. For instance, we injected 269 bugs in test *arith*, one for each of its instructions. Among those, 171 were found masked by the dynamic analysis. Our classification found that 107 of the 171 were already masked by our baseline static analysis, an additional 40 were flagged by BugMAPI as possibly masked ($<100\%$) and 24 went undetected by it (0%).

Overall, the baseline static analysis was able to detect, on average, only 62% of the masked bugs in the Power ISA tests, while BugMAPI improves the detection by an additional 15% (100% and $<100\%$ categories) for a total of 77% masked bug identification overall. Note that BugMAPI accuracy varies greatly with the contents of the test: it is very high for compute-intensive tests (first five in the plot), but less in memory-intensive tests (letter-named tests). Moreover, as we pointed out in Section 6.2, our Power ISA analysis has no memory disambiguation capabilities, which contributes to some low-accuracy results, *e.g.*, test *E*. Another type of limitation is highlighted by test *A*, where 11% of the instructions led to execution failures in dynamic analysis (*e.g.*, due to stores with misaligned address). However, BugMAPI cannot recognize these situations because of its static nature.

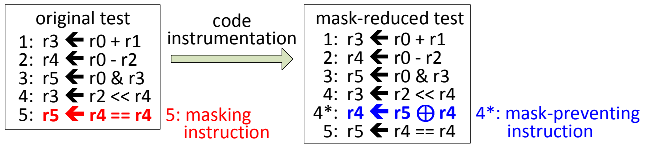
The portion of Figure 7 related to the Alpha ISA focuses on investigating the impact of memory disambiguation. The first two test-programs in this group differ only on that aspect. Note that disambiguation provides a slight improvement for the baseline static analysis, which is an improvement that BugMAPI benefits from. Finally, the last two test-programs evaluate the accuracy impact due to test length: note that the accuracy of bug-masking assessment remains stable even though the length is increased by five times.

Sources of inaccuracy. BugMAPI strives to keep the computation lightweight at the cost of some accuracy. We have identified a few sources of inaccuracy. First, our solution tracks information flow at a coarse granularity: a whole register or memory location except for a few special-purpose registers, whose fields are treated independently (*e.g.*, FPSCR in Power ISA). While a finer-granularity analysis (*e.g.*, bit-level) would lead to a more accurate estimate, it would be much more computation-heavy. Our simplifying assumptions (Section 5.2) also contribute to inaccurate estimates, because they ignore correlations between instructions, inputs and resources. We observed a minimal fraction of unmasked bugs that are reported masked by our baseline analysis; 1 out of 998 unmasked bugs for Power ISA tests, and 35 out of 4,105 unmasked bugs for Alpha ISA tests.

Finally, BugMAPI fails to recognize correlations between instructions due to test-template structures. For instance, when computing a memory address, multiple instructions are involved that are NOT independent from each other. To address this limitation, it would be possible for BugMAPI to analyze the instruction block as a single instruction with several i/o pairs, and then use those probabilities in the sequence where the block is embedded.

6.5 Discussions

Accuracy sensitivity on bug models. We performed additional experiments to measure the sensitivity of BugMAPI to other bug models, using five different types of bug manifestations: *random value overwrite*, *single-bit flip*, *binary complementation*, *missing update* and *spurious update*. Bugs in the first three types are manifested as an incorrect value in the correct target register or memory location. In the fourth type, the target maintains its previous value, ignoring its new value. Lastly, *spurious update* updates a random target other than the correct one. Figure 8 shows the accuracy for each bug model. As shown in the bottom of each bar, 38–43% of the bugs are unmasked and detected at the end of tests. Among



```
original test
1: r3 🡐 r0 + r1
2: r4 🡐 r0 - r2
3: r5 🡐 r0 & r3
4: r3 🡐 r2 << r4
5: r5 🡐 r4 == r4
```

```
mask-reduced test
1: r3 🡐 r0 + r1
2: r4 🡐 r0 - r2
3: r5 🡐 r0 & r3
4: r3 🡐 r2 << r4
4*: r4 🡐 r5 ⊕ r4
5: r5 🡐 r4 == r4
```

Figure 9: **Mask-preventing code instrumentation**. BugMAPI can identify masking instructions so that mask-preventing instructions can be inserted.

masked bugs, our baseline analysis identifies 53–81% of them, and BugMAPI adds 6–13% on top of the baseline analysis. While false positives remain minimal for the first four bug models (less than 1%), we find a 15% false positive rate with *spurious update*. This is because our analysis does not track a buggy value manifesting at a random register or memory location other than the correct target.

Applying BugMAPI to multi-threaded programs. Analyzing multi-threaded programs can be challenging because of the difficulty in characterizing interactions among threads [13]. In these programs, for instance, inter-thread data dependency may arise through memory accesses to shared memory regions. To extend our analysis to these programs, we can mark inter-thread load and store instructions, applying inter-thread analysis to them. A major challenge comes from non-deterministic execution in multi-threaded programs: inter-thread data dependency may change time to time. One solution for this challenge is to profile the frequency of each data-dependency path, then compute a weighted average across all paths. This approximation may not accurately predict bug-masking occurrences for a specific execution of a program, but it can at least estimate the overall bug-masking likelihood of the program.

7. APPLICATION: MASKING REDUCTION

The results of BugMAPI's static analysis can be used in various ways to expedite post-silicon validation. In this section, we showcase a BugMAPI's application that reduces bug-masking occurrences in random instruction tests, as illustrated in Figure 9. In this application, we first run BugMAPI to collect all instructions that possibly expunge the results of any prior instruction by overwriting its target register. We then instrument mask-preventing instructions that use the value to be overwritten as a source operand, so that any buggy value in the operand can be delivered to a propagated (non-overwritten) register.

We implement a simplistic mask-preventing code instrumentation that inserts *xor* instructions before mask-causing instructions identified by our analysis. In the random test shown in the left side of Figure 9, instruction 5 is identified as a mask-causing instruction because it expunges the results from instructions 1 and 3. We then insert an *xor* instruction using the operands of instruction 5: the target register (*r5*) and one of the source registers (*r4*), as shown in the right side of Figure 9. Note that this register selection does not require additional registers to be reserved for inserted instructions. Also note that the *xor* instruction itself does not mask any buggy value from its source operands, because its bug-masking probability is 0%. We ignore mask-causing store instructions in order to not perturb memory access patterns, limiting our mask-prevention capability to some degree. While not implemented here, a pair of load and *xor* instructions can further reduce such masking occurrences.

Table 1 shows the number of bugs that are classified as either unmasked or masked, throughout our dynamic analysis experiments. We used Alpha ISA with 100 instructions per test, and generated 100 different instruction sequences (10,000 instructions in total), using the random-value bug model and injecting a single bug to each instruction in the tests, including the instructions that we added to reduced the masking rate. As shown in the table, 57% of activated bugs in our original random tests end up being masked. However,

Table 1: **Number of bugs masked/unmasked by dynamic analysis for our enhanced tests**

test-case	number of activated bugs		
	unmasked	masked	total
original tests	4,138 (43%)	5,486 (**57%**)	9,624
mask-reduced tests	8,893 (72%)	3,384 (**28%**)	12,277

this bug masking rate is significantly lowered by our mask-reduction technique, to 28%. Note that our code instrumentation increases the length of each test, due to inserted *xor* instructions.

Our instrumentation does not eliminate all masking occurrences because of two limitations: lack of patches for mask-causing store instructions, and false negatives in our analysis, as discussed in Section 6.4.

8. CONCLUSIONS

We proposed BugMAPI, a probabilistic bug-masking analysis solution for post-silicon microprocessor validation. BugMAPI is a static information-flow analysis for test programs. It estimates the probability that bugs go undetected by the checkers at the end of the test program. It leverages a novel and highly accurate bug-masking calculation for individual instructions, and a low-cost heuristic to combine those findings into instruction sequences. Experimentally we have found that BugMAPI provides 77% of the accuracy of a dynamic simulation, at a fraction of its computation costs.

Acknowledgements. This work was supported by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Doowon Lee was also supported by an IBM PhD Fellowship.

9. REFERENCES

[1] A. Adir *et al*. Threadmill: a post-silicon exerciser for multi-threaded processors. In *Proc. DAC*, 2011.
[2] N. Binkert *et al*. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.
[3] D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.
[4] F. Fallah, S. Devadas, and K. Keutzer. OCCOM—efficient computation of observability-based code coverage metrics for functional verification. *IEEE Trans. CAD*, 20(8), 2001.
[5] S. Feng *et al*. Shoestring: probabilistic soft error reliability on the cheap. In *Proc. ASPLOS*, 2010.
[6] H. Foster. Trends in functional verification: a 2014 industry study. In *Proc. DAC*, 2015.
[7] N. Foutris *et al*. Accelerating microprocessor silicon validation by exposing ISA diversity. In *Proc. MICRO*, 2011.
[8] IBM. *Power ISA Version 2.07B*, 2015.
[9] D. Lin *et al*. Effective post-silicon validation of system-on-chips using quick error detection. *IEEE Trans. CAD*, 33(10), 2014.
[10] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proc. DATE*, 2004.
[11] A. Nahir *et al*. Post-silicon validation of the IBM POWER8 processor. In *Proc. DAC*, 2014.
[12] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. NDSS*, 2005.
[13] M. Rinard. Analysis of multithreaded programs. In *Static Analysis*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001.
[14] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1), 2003.
[15] S. Thiruvathodi and D. Yeggina. A random instruction sequence generator for ARM based systems. In *Proc. MTV*, 2014.
[16] I. Wagner and V. Bertacco. Reversi: post-silicon validation system for modern microprocessors. In *Proc. ICCD*, 2008.