

BMCFuzz: Hybrid Verification of Processors by Synergistic Integration of Bound Model Checking and Fuzzing

Shidong Shen^{1,2}, Jinyu Liu^{1,2}, Weizhi Feng^{1,2}, Fu Song^{1,2,3}, Zhilin Wu^{1,2(✉)}

¹Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

²University of Chinese Academy of Sciences, Beijing, China ³Nanjing Institute of Software Technology, Nanjing, China
shensd@ios.ac.cn, liujinyu251@mails.ucas.ac.cn, fengwz@ios.ac.cn, songfu@ios.ac.cn, wuzl@ios.ac.cn

Abstract—Modern processors are becoming increasingly complicated, making them hard to be bug-free. Bounded model checking (BMC) and coverage-guided fuzzing (CGF) are two main complementary techniques for verifying processors. BMC can exhaustively explore the state-space upto a given path-depth bound, but suffers from the infamous state-space explosion problem, thus limited to smaller bounds for realistic processor designs. CGF is efficient and scalable for verifying large-scale complex designs, but struggles with the coverage due to the difficulty in generating comprehensive and diverse seeds. To bring the best of both worlds, we propose BMCFuzz, a novel two-way hybrid verification approach that synergistically integrates BMC and CGF. Specifically, BMCFuzz alternatively switches BMC and CGF according to their performance in improving coverage, where CGF is leveraged to quickly explore the state space, detect flaws, and moreover record snapshots that are crucial valuations of all the circuit-level registers, while BMC with selected high-valuable snapshots as initial states is utilized to exhaustively explore uncovered points. Moreover, the witnesses of BMC are further used to generate seeds for CGF. This synergistic integration of BMC and CGF helps BMC alleviate the state-space explosion problem and feeds CGF with more high-quality seeds. We implement BMCFuzz as a fully open-source tool and evaluate it on three well-known open-source RISC-V processor designs (i.e., NutShell, Rocket, and BOOM). Experimental results show that BMCFuzz achieves higher coverage compared to the state-of-the-art methods and discovers three previously unknown bugs, demonstrating the potential of BMCFuzz as a powerful, open-source tool for advancing processor design and verification.

I. INTRODUCTION

Modern processors are growing increasingly complicated, incorporating advanced features, such as out-of-order execution, virtual memory, and speculative execution, to improve efficiency and enhance security. This complexity not only significantly increases the risk of design flaws but also brings a tough challenge for hardware verification. Thus, detecting design flaws early in the pre-silicon stage is essential yet very challenging for preventing costly fixes and potential security issues after tape out or even deployment [1].

Prior hardware verification approaches broadly fall into two categories: formal verification and simulation-based techniques. Formal verification techniques, such as theorem proving [2], equivalence checking [3], and model checking [4], are typically used to verify the correctness of designs, with much stronger guarantees of correctness than simulation-based techniques. Among them, model checking, as an automated formal verification technique, has been widely adopted for verifying correctness of hardware designs by converting correctness assertions and design-under-test (DUT) into a transition system and then exploring the state space of the transition system systematically and exhaustively [5]. Simulation-based techniques, such as random testing [6], hardware fuzzing [1], dynamic symbolic execution [7], and concolic execution [8], are typically used to detect design flaws and are much more efficient and scalable than formal verification techniques. Among them, hardware fuzzing, ported from software fuzzing techniques, has been shown very effective in finding

certain types of flaws, by iteratively mutating and selecting seeds based on designated feedback.

However, both model checking and hardware fuzzing still face significant challenges when applied to modern processor designs:

- **State-space explosion in model checking.** The number of states in the transition system used in model checking grows exponentially in the number of bits of the processor's registers and memory, so-called the state-space explosion problem. Thus, it is computationally infeasible to exhaustively explore all the possible states for realistic processor designs. To mitigate this issue, in practice, the state space is only systematically and exhaustively explored upto a given bound of path depth, called *bounded model checking* (BMC). Since the number of states still grows exponentially with the bound, such bound is limited for verifying modern processors. However, a small bound may be insufficient to reach critical states or explore critical behaviors, thus missing design flaws. For instance, load one word from memory when virtual memory is enabled already requires more than 30 clock cycles to complete. In summary, the larger the path-depth bound is, the more the states exhaustively explored are, but the more expensive the computation is.
- **Limited coverage in hardware fuzzing.** Fuzzing is a fundamental random-based testing technique whose effectiveness in coverage and hence flaw detection heavily relies upon the diversity and comprehensiveness of seeds. While various feedback information (e.g., coverage) has been proposed to generate comprehensive and diverse seeds that are effective in exploring common execution paths, they often struggle to exhaustively reach corner cases and complex state sequences. This limitation is particularly pronounced in the verification of modern processors with multiple privilege levels and complex control registers, where the state-space is vast and many critical behaviors require specific sequences of operations. As a result, fuzzing techniques may fail to reach critical but hard-to-reach states that could trigger important flaws.

Recognizing the complementary strengths and weaknesses of BMC and fuzzing for processor verification, recent efforts, e.g., HyPFuzz [9] and FormalFuzzer [10], have investigated hybrid approaches, aimed to improve fuzzing by BMC. While these pioneering studies have demonstrated substantial improvements in coverage and flaw detection compared to pure feedback-guided fuzzing, their integrations of BMC and fuzzing are one-way, thus *suboptimal*. Specifically, both HyPFuzz and FormalFuzzer stick to the paradigm of “BMC for fuzzing”, but not “fuzzing for BMC”. We finally remark that both HyPFuzz and FormalFuzzer use the commercial tool JasperGold for model checking, thus restricting further optimization opportunities for BMC and hindering synergistic integration of BMC and fuzzing.

Contributions. In this work, inspired by the promising improvements brought by the hybrid verification approaches HyPFuzz and Formal-

Fuzzer, we propose *BMCFuzz*, a novel two-way hybrid verification approach that synergistically integrates BMC and fuzzing.

The key challenge in synergistic integration of BMC and fuzzing lies in efficiently and thoroughly utilizing the crucial information captured during fuzzing, feeding it back to BMC as initial states, so that BMC can exhaustively explore the states that are reachable from these initial states upto a given path-depth bound, generate valuable new seeds from the witnesses, and guide fuzzing to reach even more diverse and hard-to-reach states.

To tackle this challenge, BMCFuzz makes the following technical contributions (see Section III for the details).

- 1) A concept of *snapshots* is proposed for hardware fuzzing and utilized to achieve “fuzzing for BMC”. Snapshots are valuations of all the circuit-level registers captured during fuzzing and serve as initial states for BMC. Furthermore, to mitigate the exponential blow-up of the number of snapshots, BMCFuzz chooses to *take snapshots only when the privilege level and/or any of the important bits of control and status registers changes*.
- 2) Even with the aforementioned mitigation, tens or even hundreds of snapshots can still be generated during each fuzzing. To improve the efficiency, BMCFuzz introduces a *priority-based scoring function* to select the “best” snapshot from the snapshot pool and feed it back to BMC (as an initial state). Moreover, to maintain the diversity of the selected snapshots, BMCFuzz adjusts the priorities in the scoring function dynamically.

BMCFuzz is implemented as *the first open-source and publicly available hybrid processor verification tool that combines BMC and fuzzing*, which, hopefully, will benefit the hardware-verification community and ease the further research innovation in this area. Finally, we evaluate the efficacy of BMCFuzz on three well-known open-source RISC-V processors (Nutshell [11], Rocket [12], and BOOM [13]), with respect to three widely-used coverage metrics. The experimental results show that *BMCFuzz outperforms BMC, fuzzing, and the one-way combination of them, on all three processors and in all three coverage metrics. Moreover, BMCFuzz discovers three previously unknown bugs, confirmed by the processor designers.*

Organization. The remainder of this paper is structured as follows. Section II presents an overview of our approach BMCFuzz. Section III provides more details about the various components in the BMCFuzz approach. Section IV reports the experimental evaluation of BMCFuzz. Section V discusses the related work. Section VI concludes this paper. The source code of BMCFuzz is available at: <https://github.com/iscas-versys/BMCFuzz>.

II. THE BMCFUZZ APPROACH: AN OVERVIEW

In this section, we provide an overview of our hybrid verification approach, called BMCFuzz, that synergistically integrates bounded model checking (BMC) and coverage-guided fuzzing (CGF), aimed to bring the best of both worlds.

Given a design-under-test (DUT), a golden reference model of the processor (REF), a set of initial seeds for fuzzing, and some options (e.g., the path-depth bound of bounded model checking, the size upper bound on a bundle of selected uncovered points, coverage metric), BMCFuzz systematically verifies the DUT and outputs the coverage reports and bug reports (if any bug has been discovered).

Roughly speaking, after preprocessing the DUT according to the coverage metric, BMCFuzz alternatively switches BMC and CGF according to their performance in improving the coverage. As a preparation step, BMCFuzz leverages CGF to quickly explore the state space of the DUT and detect flaws by executing a given

number of seeds (3 times of the number of initial seeds in our experiments), during which crucial valuations of all the circuit-level registers reached by fuzzing are recorded, referred to as snapshots. These snapshots after selection will be used for initializing the DUT before repeating the following BMC-CGF process: At first BMCFuzz utilizes BMC to verify the DUT and resolve the uncovered points, i.e. the coverage points that have not yet been covered by fuzzing. Once an uncovered point is resolved, the corresponding witness is produced by the model checker from which a new seed is generated. Later, BMCFuzz starts CGF and mutates the newly generated seeds to cover more points.

More specifically, as depicted in Figure 1, BMCFuzz combines BMC and CGF through a two-layer nested loop and comprises the following six main components: Preprocessor, Snapshot Manager, Initialization, Coverage-Guided Fuzzing (CGF), Bounded Model Checking (BMC), and Scheduler, each of which may comprise several sub-components. After the aforementioned preparation CGF step, BMCFuzz starts executing the nested loop, where the outer loop (blue lines in Figure 1) pops the current best snapshot from the Snapshot Pool and starts the inner loop after initializing the DUT with the snapshot, while the inner loop (red lines in Figure 1) repeats the BMC-CGF process as described above.

In the sequel, let us describe the workflow of the outer and inner loops in more details.

When the current best snapshot is provided by the Snapshot Selector, the Initialization component initializes the preprocessed DUT using the snapshot and sends it to the CGF component and the BMC component. Next, the inner loop goes as follows:

- The Point Selector first selects a bundle of uncovered points. For each selected point, the MCInstance Generator generates one cover statement (i.e., a cover property) for the point, and inserts the statement into the DUT, resulting to an instance of the model checking problem. Then the Model Checker is called to solve the model checking instance with a given path-depth bound. If the Model Checker outputs a witness (thus the selected uncovered point is resolved), then the Seed Generator generates a seed from the witness. If no seeds are generated after processing the whole bundle of uncovered points, a new bundle of uncovered points is repeatedly selected and processed as above until the bundle has been processed and some seeds are generated, or all uncovered points have been selected and processed. If the BMC component fails to resolve any uncovered points, i.e., no seeds are generated, it sends a “Terminate” signal to the Snapshot Manager component. Otherwise, all the generated seeds are sent to the CGF component and the formal rate is reported to the Scheduler component.
- The seeds received from the BMC component are stored in the Seed Corpus. To generate more effective and diverse seeds, all the seeds are mutated by the Mutator in the CGF component according to their coverage improvement, where the coverage improvement is calculated when the Simulator executes the instruction sequence in the seeds on the DUT. Note that when executing the instruction sequence in each seed, the same initialized DUT as provided to the BMC component is executed. Furthermore, during CGF, snapshots are recorded and sent to the Snapshot Pool, the coverage information is updated and shared with the BMC component, and the fuzzing rate is updated and reported to the Scheduler component each time when the Simulator is called.
- To determine when switching from the CGF component to the BMC component, we use the fuzzing rate and formal rate. If the fuzzing rate is less than the formal rate, then the Scheduler component decides that it might be less beneficial to continue

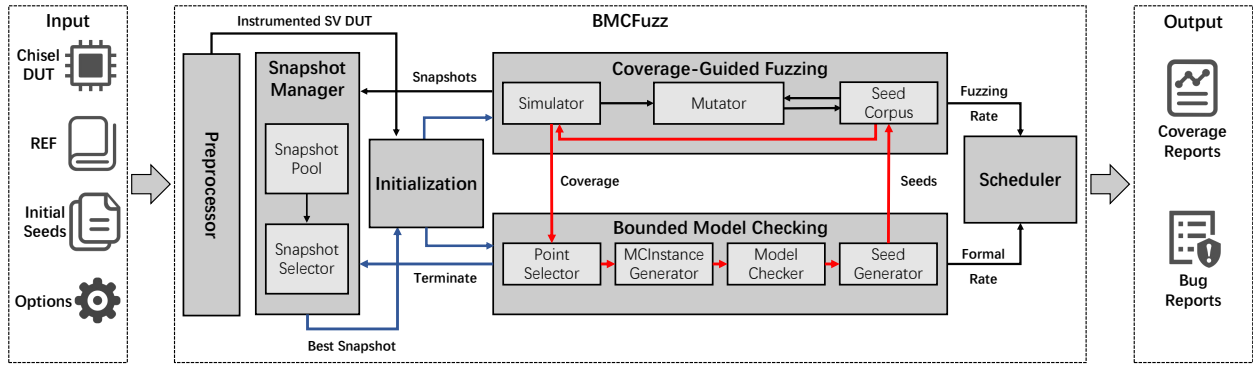


Fig. 1: The BMCFuzz approach

running the CGF, thus it stops the CGF and starts a new round of the BMC-CGF process (i.e. the body of the inner loop).

Note that during the aforementioned CGF process, the snapshots are taken and stored into the Snapshot Pool, which will be used in the outer loop. The outer loop is controlled by the Snapshot Manager. Once receiving a “Terminate” signal from the BMC component, the Snapshot Manager component stops the inner loop, and the Snapshot Selector utilizes a carefully designed priority-based snapshot-scoring function to select the current best snapshot from the Snapshot Pool. This best snapshot is used to initialize the preprocessed DUT again and the inner loop is restarted using the newly initialized DUT.

III. THE BMCFUZZ APPROACH: A CLOSER LOOK

In this section, we have a closer look at the main components in the BMCFuzz approach, except for the Schedule component which has been clearly explained in Section II. We will demonstrate the effectiveness of BMCFuzz for verifying RISC-V processor designs in Chisel [14], an open-source hardware construction language embedded in Scala, thus some details may be RISC-V specific. We shall zoom in on the concept of snapshots and the Snapshot Manager component, before diving into the other components.

A. Snapshots and Snapshot Manager

Snapshots. During the CGF process, the Simulator should record snapshots, i.e., valuations of all the (circuit-level) registers, including not only the architectural registers, such as general-purpose registers, control and status registers (CSRs), but also various internal microarchitectural registers of the processor. These snapshots after selection will be used to initialize the DUT for both BMC and CGF. However, there are too many such valuations reached during CGF and it is virtually impossible to record all of them. To mitigate this issue, the Simulator is designed to record only the valuations of all the registers when the privilege level and/or any value of the important bits of certain CSRs changes, based on the practical experience of ProcessorFuzz [15]. ProcessorFuzz showed that the privilege level and some bits of certain CSRs play a significant role in controlling processor behavior switching, thus monitored the important fields of the following eight CSRs: `mstatus`, `mcause`, `scause`, `medeleg`, `mcounteren`, `scounteren`, `frm`, and `fflags`. However, we found that `mcause`, `scause`, `mcounteren`, and `scounteren` are less important, because (1) `mstatus` also changes when either `mcause` or `scause` changes; (2) `mcounteren` and `scounteren` only control the availability of the hardware performance-monitoring counters for supervisor and user mode, respectively. While `frm` and `fflags` are important, they are related to floating-point operations with which processors go beyond the capability of current model checkers. Thus, in this work, besides the privilege level, we consider

three CSRs: `mstatus`, `medeleg`, and `satp`, where `satp` controls supervisor-mode address translation and protection and thus is important when the DUT supports virtual memory. Specifically, we monitor: the `MXR`, `SUM`, `TSR`, `TW`, `TVM`, `MPP`, `MPRV`, `SPP`, `MPIE`, `SPIE`, `MIE` and `SIE` fields of `mstatus`, all the fields of `medeleg`, and the `MODE` field of `satp`. Details of these fields can be found in [16]. Thus, when the privilege level (referred to as `PLevel`) or a monitored field changes at a clock cycle, the updated valuation of all the registers is recorded as a snapshot by the Simulator. Moreover, the snapshot is attached with a set of tags $\text{Tags}(s)$, explaining why the snapshot s is recorded. For instance, $\text{Tags}(s) = \{\text{PLevel}, \text{MXR}\}$ means that both the privilege level and the `MXR` field are updated before recording the snapshot s . We note that more fields and more CSRs could be added and monitored in a similar way.

Snapshot Manager. The Snapshot manager component comprises two sub-components, namely, Snapshot Pool and Snapshot Selector. In the Snapshot Selector, we introduce a dynamic priority-based snapshot-scoring function `Score` for choosing the best snapshot from the Snapshot Pool by utilizing their tags.

Recall that a snapshot s is recorded because the monitored fields in $\text{Tags}(s)$ (may include the privilege level) change. We partition all the monitored fields into five groups, each of which is associated with a priority weight. A snapshot is scored using the priority weights of the groups that intersects with the snapshot’s tags. To be diverse, all the priority weights will be dynamically adjusted by a decay factor when they are involved in endorsing the best snapshot, while the other weights remain the same.

Specifically, all the monitored fields are partitioned as follows according to five selection criteria for RISC-V processors:

C1: Privilege level. According to the RISC-V ISA, there are three privilege levels: Machine (M), Supervisor (S), and User (U). This privilege level mechanism effectively isolates and protects system resources, ensuring that only authorized code can perform specific operations, thereby enhancing system security and stability. Consequently, privilege levels significantly impact the processor’s behaviors. Thus, we define group $G_1 = \{\text{PLevel}\}$.

C2: Virtual memory. The RISC-V ISA supports virtual memory to facilitate efficient memory management and process isolation. For instance, Sv39 provides a 39-bit virtual address space. Enabling/disabling virtual memory significantly impacts the processor’s behaviors, particularly affecting the coverage of the Translation Lookaside Buffer (TLB). Thus, we define group $G_2 = \{\text{MXR}, \text{SUM}, \text{MPRV}, \text{MODE}\}$, which is closely related to the setup of virtual memory, virtual address accesses and translation.

C3: Specific illegal-instruction exception. Some operations require specific values of the fields `TVM`, `TW`, `TSR`, otherwise an illegal-

instruction exception raises. Indeed, an illegal-instruction exception will raise if any of the following cases happens: (1) executing `SFENCE.VMA` and `SINVAL.VMA`, or read/write the `satp` register when executing in the Supervisor mode and `TVM = 1`; (2) an execution of `WFI` in any less-privileged mode, does not complete within an implementation-specific, bounded time limit when `TM = 1`; and (3) executing `SRET` when executing in the Supervisor mode and `TSR = 1`. Thus, we define group $G_3 = \{\text{TVM}, \text{TW}, \text{TSR}\}$.

C4: Trap. We define group $G_4 = \{\text{MPP}, \text{SPP}, \text{MPIE}, \text{SPIE}, \text{MIE}, \text{SIE}\}$, that are the fields of the `mstatus` register and their values are closely related to trap (i.e., interruption or exception).

C5: Exception delegation. The value of the `medelegs` register determines the delegation for handling the exceptions. Thus, changing fields of the `medelegs` register indeed configures the way of delegations of exception handling. Thus, we define group G_5 as the set of all fields of the `medelegs` register.

We denote by W_1, \dots, W_5 the priority weights of the groups G_1, \dots, G_5 , respectively. Given a pool S of snapshots, the best snapshot of S is selected as the snapshot s such that $\text{Score}(s)$ is the greatest among those snapshots in S , where

$$\text{Score}(s) = \sum \{W_i \mid \text{Tags}(s) \cap G_i \neq \emptyset\}.$$

Note that if there are multiple snapshots having the same maximal score, the snapshot that is recorded first is selected.

As aforementioned, once the best snapshot s is popped from the pool S , all the priority weights that are involved in endorsing the best snapshot are dynamically adjusted by a decay factor. Let D be the decay factor. The priority weight W_i is adjusted as follows:

$$W_i := \begin{cases} W_i \times D, & \text{if } \text{Tags}(s) \cap G_i \neq \emptyset; \\ W_i, & \text{otherwise.} \end{cases}$$

After some preliminary experiments, we set $W_1 = 2^3, W_2 = 2^5, W_3 = 2^4, W_4 = 2^3, W_5 = 2^2$, and $D = 2^{-1}$, which turns out to perform well in our experiments. Fine-partitioning the fields and fine-tuning priority weights are left as future work.

B. Preprocessor

The Preprocessor component first compiles the given Chisel DUT into FIRRTL (Flexible Intermediate Representation for RTL) by utilizing the Scala CLI with Chisel library and Java Virtual Machine (JVM). During this compilation, the DUT is instrumented according to the chosen coverage metric and monitored fields of registers via an instrumentation pass, i.e., inserting statements into proper positions so that these statements will emit coverage information and snapshots when executing. Next, by utilizing the FIRRTL compiler Firtool, the instrumented DUT in FIRRTL is further compiled into SystemVerilog (SV) which can be processed by both BMC and fuzzing.

According to [17]–[19], we focus on the following three mainstream coverage metrics:

- Register-toggle (Reg-tog) coverage: whether each bit of each register in the DUT is flipped or not during simulation;
- Branch coverage: whether the different paths of each branch statement are covered or not during simulation;
- Multiplexer-toggle (Mux-tog) coverage: whether the condition signal of each multiplexer is flipped or not during simulation.

C. Initialization

Given a snapshot and a preprocessed DUT, the Initialization component initializes the preprocessed DUT and sends the initialized DUT to the CGF component and the BMC component. The Initialization component utilizes the snapshot to generate initialization statements for all the circuit-level registers and inserts these statements

into the preprocessed DUT. The cache and TLB in the initialized DUT may not be empty, leading to a mismatch between the initialized DUT and the reference model during fuzzing. To maintain consistency, all the valid bits of the cache and TLB lines are invalidated in the DUT. Note that the Initialization component just resets the DUT at the preparation CGF step where no snapshot is available.

D. Coverage-Guided Fuzzing

In general, any coverage-guided hardware fuzzer can be adapted for BMCFuzz. In our implementation, we adapt the recent promising fuzzer, PATHFUZZ [20], an extension of the coverage-guided hardware fuzzer, XFUZZ [17]. XFUZZ is built on the state-of-the-art modular software fuzzer LibAFL [21] and the modern co-simulation framework DiffTest [22] for RISC-V processors. In XFUZZ, seeds are interpreted as a linear memory with contents organized according to the address order which is misaligned with the processor execution. PATHFUZZ overcomes this issue by extending XFUZZ with a footprint memory that only chronologically records contents in an ordered sequence aligned with the processor execution, thus improves fuzzing capability. Thus, BMCFuzz uses DiffTest as the RISC-V processor co-simulation framework and the mutation engine of LibAFL as the Mutator, and either linear or footprint memory to represent seeds, where the Mutator determines whether to mutate a seed or not based on its coverage improvement and mutates the seed by performing various data manipulation operations such as bit-flip, byte-flip, clone, and swap. The coverage improvement of a seed is calculated as the number of newly covered points by executing this seed.

Below, we discuss the details of the Simulator and fuzzing rate.

Simulator. Given the initialized DUT in SV, a golden reference model (REF), and a seed, the Simulator first executes all the uncommitted instructions in the initialized DUT by utilizing the tool Verilator [23]. After these uncommitted instructions have been executed and committed, the Simulator synchronizes the latest state of the DUT with the REF, co-simulates the DUT and REF using the seed during which the behaviors (e.g., CSRs) of the DUT and REF are compared using the DiffTest framework. DiffTest allows users to define behaviors to be compared, such as load/store event, exceptions and interrupts. Currently, we use the C++ implementation of the RISC-V ISA simulator, Spike [24], as the golden reference model and compare the DUT with Spike using the behaviors defined in DiffTest (cf. [25]), the same as PATHFUZZ and XFUZZ. If any discrepancy of the considered behaviors between the DUT and the REF is found, a potential bug is identified and reported. Besides recording snapshots, the Simulator also updates the coverage information (i.e., whether coverage points are covered or not) and the fuzzing rate during the co-simulation for each seed.

Fuzzing rate. The fuzzing rate is calculated as the average of coverage rates of the latest (upto) 100 executions (one execution per seed) in the same invocation of the CGF component (i.e., switched from the BMC component), where the coverage rate of one execution (thus one seed) is calculated as the number of newly covered points per second in the execution. The updated fuzzing rate is sent to the Scheduler component.

E. Bounded Model Checking

We first discuss the four key sub-components in the BMC components and then define the formal rate.

Point Selector. According to coverage information from the CGF component, the Point Selector iteratively selects a bundle of uncovered points until all the uncovered points in the same bundle have been verified by the Model Checker and some of them are

resolved, or all uncovered points have been selected and processed. To select a bundle of uncovered points such that the seeds generated from the witnesses reported by the Model Checker are diverse and comprehensive (i.e., bringing higher coverage improvements during CGF), inspired by the MaxUncovered strategy of HyPFuzz, the Point Selector sorts all the submodules in the DUT according to their number of uncovered points. Intuitively, the submodule with the most uncovered points is the least-covered region, thus CGF using seeds generated for covering this region is more likely bringing higher coverage improvements. Thus, the Point Selector randomly selects a bundle of uncovered points from the submodules according to their descending order, and sends them to the MCInstance Generator. To avoid redundant model checking instances, for the same initialized DUT, each uncovered point is selected at most once by the Point Selector.

MCInstance Generator. Before generating a model checking instance, the MCInstance Generator first abstracts the memory of the DUT to facilitate the subsequent SAT-based bit-level BMC. After that, for each uncovered point that is received from the Point Selector, the MCInstance Generator generates and inserts a corresponding cover statement (i.e., a cover property in SystemVerilogAssertions) into the resulting DUT. Remark that memory abstraction is used exclusively during BMC rather CGF. Below, we describe our memory abstraction.

First, directly ignoring the entire memory of the DUT and giving an unrestricted value for each load/fetch/write operation, would violate memory consistency, because two consecutive load/fetch/writes of the same memory address should give the same value unless there is an overwriting between the two load/fetch/writes. To tackle this issue, we design a cache-like abstract memory, aimed to maximize the path-depth bound in bounded model checking. Our memory abstraction follows a key principle: the first load/fetch/write of a memory address returns an arbitrary valid value, while the subsequent load/fetch/write of the same memory address returns the same value as the first load/fetch/write if there is no overwriting, otherwise the overwritten value is loaded/fetched/written. This principle enforces the memory consistency and meanwhile effectively reduces the state space in bounded model checking.

Let us exemplify the memory abstraction using the Rocket processor [12]. The full memory of the Rocket SOC [12] has a 2GiB address space (equivalent to 2^{34} bits), so that model checking using this memory can only explore 8 instructions in 20 minutes. Consider an abstract memory that comprises 16 entries of 64-bits, thus capable of storing up to 32 instructions or 128-bytes data. This abstract memory requires only 1504 bits in total: (64 bits for data + 28 bits for tag + and 1 bit for validity) \times 16 cache lines + 16 bits to record the place to refill data when missing, achieving a size reduction by a factor of approximately 1.14×10^7 . Using this abstract memory, bounded model checking of Rocket can explore 16 instructions in 20 minutes, which is two times of that of the full memory.

Model Checker. For bounded model checking, there are two choices: SAT-based (bit-level) BMC and SMT-based (word-level) BMC. We do not consider BDD-based BMC, since it is significantly less scalable than SAT/SMT-based BMC.

- **SAT-based BMC** converts high-level RTL designs into low-level netlists in which all signals are Boolean. AIG (And-Inverter Graph) is commonly used to model netlists on which model checking is performed by harnessing an SAT solver.
- **SMT-based BMC** directly utilizes bit-vector arithmetic to model high-level RTL designs and array theory to model the memory. BTOR2 and SMTLIB2 are the two most common ways for

modeling high-level RTL designs on which model checking is performed by harnessing an SMT solver.

An SMT solver can either transform an SMT instance with fixed lengths of arrays and bit-vectors into an SAT one via bit-blasting or directly solve an SMT instance using a decision procedure where SAT solving may be internally used. However, bit-blasting may be computationally infeasible when large arrays are involved, e.g., the SMT-based BMC instance of a processor with full memory.

Prior studies have shown that there is a trade-off between SAT-based BMC and SMT-based BMC [26]: SAT-based BMC usually performs far superiorly on bit-level instances when applied after successfully bit-blasting the word-level instances, while SMT-based BMC is more powerful and suitable for preserving high-level information and verifying processor designs with full memory. To utilize efficient SAT-based BMC for verifying processor designs that commonly have a large memory, we apply the aforementioned memory abstraction. As a result, a witness is a sequence of values loaded/fetched from or written to the abstract memory when traversing the transition system to reach the uncovered point, where a placeholder is inserted if no value is loaded/fetched/written at a clock cycle.

To see the effectiveness of our memory abstraction, we use the widely-used open-source EDA tool SymbiYosys [27] to verify a Reg-tog cover point (`cover: TLTOAXI4.counter_0[0]`) of the Rocket SOC using both the SMT-based BMC and SAT-based BMC. Since the SAT-based BMC of SymbiYosys is very slow due to the outdated SAT solver, we build a model checker, named SymbiYosys+r1C3, by extending SymbiYosys with the latest Rust implementation of BMC provided by r1C3 [28], the winner of multiple tracks in HWMCC-2024 [29]. The SAT-based BMC takes 2 minutes to solve the problem, while the SMT-based BMC takes 20 minutes with the fastest SMT solver, Bitwuzla [30]. This result indicates that SAT-based BMC with our memory abstraction is much more efficient than SMT-based BMC, thus can explore much deeper states within the same time budget. Thus, the Model Checker invokes SymbiYosys+r1C3 to solve model checking instances in parallel where memory is abstracted in MCInstance Generator.

Seed Generator. When the Model Checker produces a witness (thus the selected uncovered point is resolved), the Seed Generator generates a seed from the witness. However, there is a representation gap between the witnesses and the seed: the witness generated by model checking the DUT with an abstract memory is a sequence of placeholders and/or values loaded/fetched/written from the abstract memory when traversing the transition system to reach the uncovered point, while the seed required by the fuzzer is either linear memory or footprint memory. To fill this gap, the Seed Generator invokes a revised version of the Simulator which can execute the initialized DUT using the witness as the input and reconstruct a linear/footprint memory. Specifically, the Simulator is revised as follows: (1) when the execution attempts to load/fetch a value from some address in the linear/footprint memory at a clock cycle, the corresponding value in the witness at the same clock cycle is fed to the execution and moreover is stored into the linear/footprint memory at that address, and (2) when the execution attempts to write a value to some address, the linear/footprint memory is not updated. The reconstructed linear/footprint memory is the desired seed, thus is sent to the CGF component which will add it into the Seed Corpus.

Formal rate. Different from the fuzzing rate which is updated for each seed and reset for each invocation of the CGF component, the formal rate is updated for each invocation of the BMC component and reset for each initialized DUT using a snapshot. Specifically, the formal rate is calculated as the moving average of the coverage rates

TABLE I: Characteristics of three RISC-V processors, where BP denotes Branch Prediction, TLB denotes Translation Lookaside Buffer.

Name	#LoC	Pipeline	Single Issue	OoO	Sv39	TLB	Privilege	BP	Cache
NutShell	7,220	9 stages	✓	✗	✓	✓	M, S, U	✓	L1
Rocket	11,542	5 stages	✓	✗	✓	✓	M, S, U	✓	L1
BOOM	13,123	7 stages	✓	✓	✓	✓	M, S, U	✗	L1

of all invocations of the BMC component for the same initialized DUT, where the coverage rate of one invocation of the BMC component is the number of resolved uncovered points per second in this invocation. For instance, suppose the formal rate is r after k invocations of the BMC component for an initialized DUT, and the $(k + 1)$ -th invocation resolves n uncovered points using t seconds. Then, the updated formal rate is $\frac{r \times k + n/t}{k+1}$. Note that the up-to-date formal rate is sent to the Scheduler component when switching to the CGF component. Intuitively, the formal rate estimates the trend of the BMC component's ability for resolving uncovered points.

IV. EVALUATION

In this section, we evaluate the efficacy of BMCFuzz.

A. Well-known Open-source Benchmarks

We use three well-known open-source RISC-V processors implemented in the Chisel language: NutShell [11], Rocket [12], and BOOM [13]. They all support Sv39 virtual memory, and RV64 I, M, A, C, Zicsr, Zifencei, and Zihpm instruction extensions except that NutShell does not support Zihpm. Other characteristics of these processors¹ can be found in Table I. These processors vary in size (number of lines of code), pipeline stage (5, 7, 9 respectively), and execution mode (in-order, out-of-order, abbreviated as OoO), thus constituting a reasonable benchmark set for evaluating the efficacy of BMCFuzz. We note that the Rocket processor considered in this paper is an extension of the Rocket core used in PATHFUZZ [20] and HyPFuzz [9], in which Sv39 and TLB are excluded.

B. SOTA Open-source Baselines

We aim to compare BMCFuzz with the state-of-the-art (SOTA) open-source tools. We consider three SOTA baselines: BMC tool SymbiYosys+riC3, DiffTest-based CGF tool PATHFUZZ [20], and hybrid verification tool HyPFuzz [9] that combines BMC and CGF. We do not consider non-DiffTest-based CGF tools since the three benchmarking RISC-V processors are all implemented in Chisel and DiffTest [25] is a widely-used testing framework for Chisel.

We take SymbiYosys+riC3 as the SOTA open-source BMC tool, which is used as the Model Checker in BMCFuzz. SymbiYosys is a widely-used open-source hardware formal verification front end supporting both SystemVerilog and Verilog, where various model checking engines can be used as back ends. The model checker riC3 is taken as the best open-source hardware model checker since it won both the bit-level and word-level tracks in HWMCC'24, the latest Hardware Model Checking Competition.

We take PATHFUZZ as the SOTA open-source DiffTest-based CGF tool. PATHFUZZ proposes the concept of footprint memory in DAC 2024 and replaces the linear memory used in XFUZZ [17] by the footprint memory to enhance the performance of XFUZZ. XFUZZ, built on the state-of-the-art modular software fuzzer LibAFL [21], is an open-source CGF framework for verifying RISC-V processor Chisel designs. Thus, PATHFUZZ is considered as the SOTA DiffTest-based CGF tool for verifying RISC-V processor Chisel designs.

¹Some of the characteristics are configurable and the characteristics given here are those that will be used in the evaluation.

TABLE II: Time limit and BMC step bound

Processor	Coverage	Timeout(h)	BMC timeout (h)	BMC step
NutShell	Reg-tog	24	2	90
	Branch	12	2	90
	Mux-tog	12	2	90
Rocket	Reg-tog	24	3	75
	Branch	12	3	75
	Mux-tog	12	3	75
BOOM	Reg-tog	48	4	75
	Branch	24	4	75
	Mux-tog	24	4	75

However, HyPFuzz, the SOTA hybrid verification tool that combines BMC and CGF, is not publicly available (the same as the another hybrid verification tool FormalFuzzer [10]). Thus, we implement HyPFuzz*, that is, the inner loop of BMCFuzz, but both the DUT and reference model are initialized by reset instead of snapshots, indicating that snapshots and initial seeds in the BMCFuzz are not used in HyPFuzz*. HyPFuzz* can be seen as an extended open-source implementation of HyPFuzz with its best strategies. As far as we can see, HyPFuzz does not support the footprint memory, which was proposed by PATHFUZZ, later than HyPFuzz, while HyPFuzz* supports both linear memory and footprint memory. HyPFuzz* is taken as a baseline for demonstrating the performance gain of the two-way synergetic integration of BMC and CGF, compared to the one-way integration of BMC and CGF in HyPFuzz.

Finally, we found that the behavior of PATHFUZZ on BOOM is abnormal when the footprint memory is used². Thus, linear memory is used in all the tools BMCFuzz, HyPFuzz*, and PATHFUZZ on BOOM, while footprint memory is used on NutShell and Rocket.

C. Experimental Setup

All experiments are conducted on an Ubuntu 22.04 LTS system with 2048 GiB RAM and 256 Intel(R) Xeon Platinum 8153 CPUs (2.00 GHz). In the experiments, we choose Spike [24] as the golden reference model. Moreover, in riC3, the SAT solver Cadical [31] is used. Finally, in BMC (SymbiYosys+riC3), the path-depth bound is set to 90 for NutShell and 75 for both Rocket and BOOM.

The initial seeds are required in running BMCFuzz, PATHFUZZ, and XFUZZ. Moreover, in the preparation CGF step, the initial seeds are used to generate snapshots, which will be used for initializing the DUT. Thus, when collecting the initial seeds, we extend the riscv-dv seeds [32] used by PATHFUZZ with the riscv-tests [33], since the seeds in riscv-tests contain many privilege-level switches as well as the operations of opening the virtual memory, thus snapshots of higher quality can be generated, compared to the seeds in riscv-dv. Finally, we obtain 1290 seeds (more specifically, 140 seeds from riscv-tests and 1150 seeds from riscv-dv), as the set of initial seeds. These seeds are represented as footprint memory for NutShell and Rocket, while they are represented as linear memory for BOOM.

The configuration of the time limits and BMC path-depth bounds can be found in Table II. Remark that the total time limits and the BMC time limits can be increased when the processor sizes grow, that is, in the order of Nutshell, Rocket and BOOM, while the BMC path-depth bounds can be decreased simultaneously. Moreover, the total time limits can vary for different coverage metrics, since the number of coverage points can be quite different for different metrics.

²The abnormal behavior is due to the fact that the cache in BOOM is non-blocking. As a result, in PathFuz, the footprint memory generated during the simulation of BOOM is different from that generated during the simulation of the reference model, resulting to many false positives.

TABLE III: Experimental results, where $x(a, b)$ in the BMCFuzz column denotes that the coverage of BMCFuzz is $x\%$, which is $a\%$ (resp. $b\%$) more than that of PATHFUZZ (resp. HyPFuzz*), and † denotes we use linear memory instead of footprint memory.

DUT	Coverage	Coverage Points	SymbiYosys+riC3(%)	PATHFUZZ (%)	HyPFuzz*(%)	BMCFuzz(%)
NutShell	Reg-tog	13,003	31.77	84.77	89.38	89.78(5.01, 0.40)
	Branch	1,939	69.73	94.22	95.36	95.93(1.71, 0.57)
	Mux-tog	1,349	48.78	93.63	94.89	95.26(1.63, 0.37)
Rocket	Reg-tog	8,744	24.10	85.58	78.95	86.78(1.20, 7.83)
	Branch	1,358	37.41	70.99	68.19	71.06(0.07, 2.87)
	Mux-tog	1,306	32.85	76.88	73.12	77.41(0.53, 4.29)
BOOM	Reg-tog	19,229	22.19 †	83.91 †	72.63 †	84.71(0.80, 12.08)†
	Branch	12,966	58.37 †	92.22 †	78.52 †	93.80(1.58, 15.28)†
	Mux-tog	4,629	41.89 †	83.43 †	80.19 †	88.51(5.08, 8.32)†

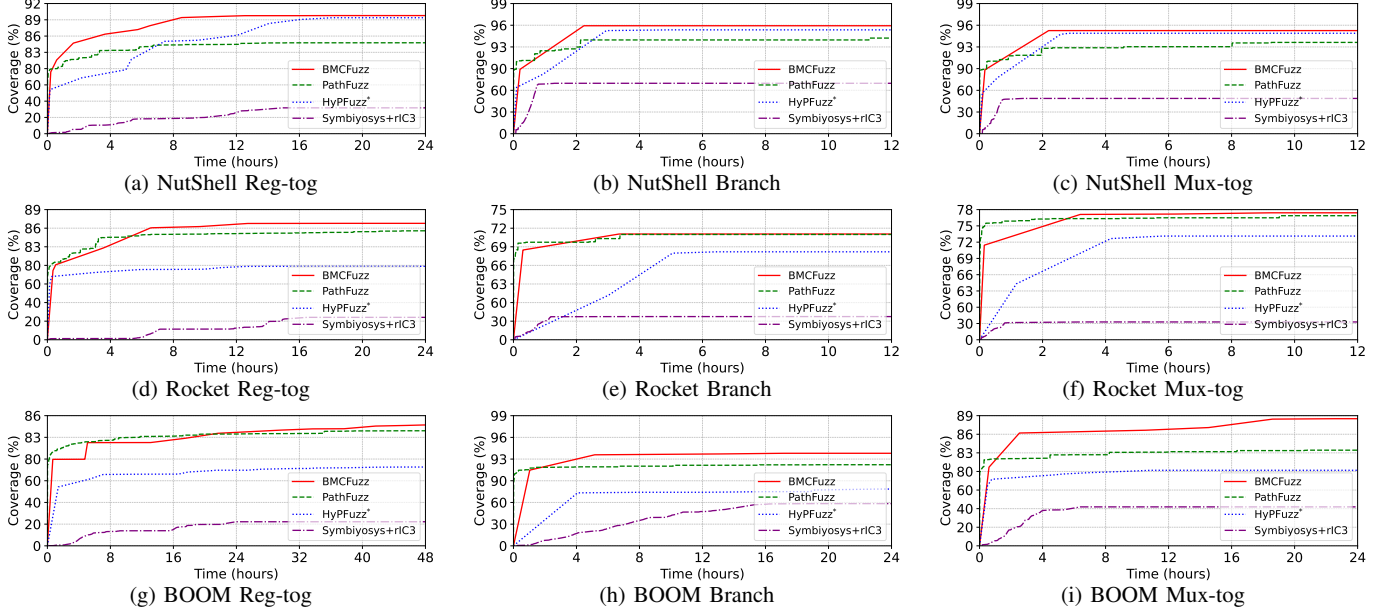


Fig. 2: Coverage curves of different tools

D. Experimental Results

1) *Coverage comparison*: Table III reports the coverage results after timeout. Obviously, **BMCFuzz achieves the best coverage in all the combinations of three processors and three coverage metrics**. In the sequel, we highlight some of the results.

- The SOTA pure model checker SymbiYosys+riC3 has very low coverage attributed to the computational cost of model checking.
- Compared with PATHFUZZ, BMCFuzz achieves 5.01% more Reg-tog coverage, 1.71% more branch coverage, and 1.63% more Mux-tog coverage on NutShell.
- Compared with HyPFuzz*, BMCFuzz achieves 12.08% more Reg-tog coverage, 15.28% more branch coverage, and 8.32% more Mux-tog coverage on BOOM.
- Compared with HyPFuzz*, the improvement of BMCFuzz in all three coverage metrics becomes much more significant with increase of the processor size, from NutShell with 7,220 #LoC and Rocket with 11,542 #LoC to BOOM with 13,123 #LoC. This difference is attributed to the fact that model checking larger processor is much harder so that HyPFuzz* (one-way combination of BMC and CGF) achieves a limited coverage, even much worse than PATHFUZZ on the largest processor BOOM, while BMCFuzz (two-way synergistic integration of BMC and CGF) is more effective.
- The coverage improvement of BMCFuzz also varies on processors, when compared with PATHFUZZ and HyPFuzz*. For instance, when compared to PATHFUZZ, the performance of BMCFuzz is better on Nutshell and BOOM, in all three coverage metrics (except for the Reg-tog coverage on BOOM). On the other hand, when

compared to HyPFuzz*, the performance of BMCFuzz is better on Rocket and BOOM, in all three coverage metrics.

Figure 2 depicts coverage trends along with execution time. We can see that BMCFuzz surpasses all the other tools already in around 3 hours (except on Rocket, in the Reg-tog coverage, which is around 5 hours) and keeps the lead afterwards. This shows that **BMCFuzz not only achieves the best coverage in all three coverage metrics and on all the three processors, but also achieves the best coverage in a fast speed**.

2) *Bug finding*: During the evaluation, **BMCFuzz discovers three bugs on Nutshell and one bug on BOOM** (cf. Table IV), among which three are newly discovered. Moreover, the three bugs discovered on Nutshell have been confirmed by the designers.

V. RELATED WORK

In this section, we discuss hardware fuzzing, hardware model checking, and hybrid verification that are closely related to this work.

1) *Hardware Fuzzing*: Inspired by the success of fuzzing techniques in software bug detection, hardware fuzzing was studied recently to increase the exploration of design spaces and accelerate the detection of security flaws, cf. [1] for a review. RFUZZ [18] is one of the first hardware fuzzers which uses the mux-tog coverage as feedback to mutate seeds and improve code coverage, a key point distinguishing hardware fuzzing from random testing. Subsequently, more coverage metrics are proposed and used as feedback, aimed to generate high-quality seeds. For instance, DiFUZZRTL [34] uses register coverage metric; TheHuzz [35] supports finite-state machine, branch, toggle, and conditional coverage metrics; XFUZZ [17] and

TABLE IV: Bugs discovered in the evaluation

Processor	Bug ID	Bug Description	New	Confirmed by designers
NutShell	N1	The reserved D, A, and U bits of non-leaf page table entries (PTEs) may not be zeroed, inconsistent with the RISC-V specification and the reference model Spike	✓	✓
NutShell	N2	When the Sv39 virtual memory is enabled, a TLB exception causes NutShell to hang due to a hardcoded condition that disallows the required exception propagation	✓	✓
NutShell	N3	Illegal read/write mask of the control and status register <code>medeleg</code> which determines the delegation of exception handling (a wrong mask <code>0xbfff</code> is used instead of the correct one <code>0xb3ff</code>)	✗	✓
BOOM	B1	When Svnoput is not implemented and the reserved 63-th bit of PTEs is not zeroed, the pagefault exception did not raise, inconsistent with the RISC-V specification and the reference model Spike	✓	✗

PATHFUZZ [20] support various functional coverage metrics (e.g., instruction coverage) and code coverage metrics (e.g., mux, control register); and ProcessorFuzz [15] uses CSRs and ISA-transition coverage metrics. Other approaches for improving quality of seeds include: intricate program generation [36], [37], large language models [38], and ISA-specific optimizations [39], [40].

Hardware fuzzing can be implemented in different ways: (1) adapting software fuzzers for hardware (e.g., RFUZZ, XFUZZ, PATHFUZZ), (2) fuzzing hardware as a software (e.g., Trippel et al. [41]) that translates the DUT into a software version on which a software fuzzer can be directly harnessed; and (3) fuzzing hardware as hardware (e.g., TheHuzz, ProcessorFuzz, DIFUZZRTL [34]) that integrates hardware fuzzing into conventional hardware design and verification workflows. To our knowledge, only PATHFUZZ supports the promising footprint memory while the others use linear memory.

Despite substantial improvements over random testing and random regression, the coverage of current hardware fuzzers is still far from the industry standards [9]. While any promising coverage-guided fuzzer can be adapted to implement the CGF component in BMCFuzz, to demonstrate the efficacy of BMCFuzz, we adapt the recent promising fuzzer PATHFUZZ, supporting two memory models (linear memory and footprint memory) and three coverage metrics (register-toggle, branch and mux-tog).

2) *Hardware Model Checking*: Model checking plays a vital role in hardware verification [5], as evidenced by three widely-used commercial formal verification EDA tools: Questa [42], VC Formal [43], and JasperGold [44], and the famous open-source EDA tool SymbiYosys. These tools can exhaustively verify whether a DUT satisfies specified properties. Besides these general EDA tools, there are dedicated model checkers (e.g., OneSpin [45], FormalISA [46], χ RVFormal [4] and riscv-formal [47]) for verifying compliance of RISC-V processors w.r.t. the RISC-V specification.

Though these tools can exhaustively verify a DUT, they alone cannot verify an entire modern processor design due to the state-space explosion problem. Furthermore, writing properties in SVA often requires manual effort and expert knowledge of the DUT, which is error-prone and time-consuming. While any promising bounded model checker can be leveraged to implement our BMC component in BMCFuzz, to demonstrate the efficacy of BMCFuzz, we extend SymbiYosys with the BMC engine of rIC3, the winner of multiple tracks in HWMCC-2024.

We note that there are other specific formal verification techniques which have been shown effective in detecting subtle vulnerabilities, such as side-channel leakage (e.g., [48]) and information leakage (e.g., [49]). These are orthogonal to our work.

3) *Hybrid verification*: Recent efforts, HyPFuzz [9], FormalFuzzer [10] and BMC+Fuzz [50], proposed hybrid verification approaches by combining BMC and CGF. These pioneering studies have demonstrated substantial improvements over pure feedback-guided fuzzing, where HyPFuzz and BMC+Fuzz use coverages as feedback while FormalFuzzer uses cost values as feedback. However, their integrations of BMC and fuzzing are one-way, i.e., improving

fuzzing by BMC, thus are suboptimal. Specifically, HyPFuzz invokes BMC multiple times to resolve uncovered points and uses witnesses obtained from BMC to generate seeds. However, its multiple invocations of BMC always use the same fixed initial states to explore all the uncovered points and no information obtained from fuzzing is utilized in BMC. As a result, the BMC component of HyPFuzz fails to explore the uncovered states that go beyond the given depth bound. FormalFuzzer only applies BMC as a preprocessing step to reduce the input space and the seed mutation space for subsequent cost-guided fuzzing. Thus, no information from fuzzing is utilized in BMC. Furthermore, the assertions supported by FormalFuzzer are limited to five templates, and seed mutation space should be manually determined by verification engineers based on the results of BMC, instruction set architecture and design specification. BMC+Fuzz targets software programs and utilizes BMC to generate seeds for CGF, which is only a one-way combination of BMC and fuzzing, while BMCFuzz in this work is a two-way combination.

There are also methods that combine formal verification and simulation such as [51], [52]. The combination of BMC and simulation is easier than that of BMC and CGF since both BMC and simulation start from states (or sets of states), while CGF takes seeds as inputs.

In contrast to them, BMCFuzz is a novel two-way hybrid verification approach that synergistically integrates BMC and fuzzing: not only improving fuzzing by BMC, but only improving BMC for resolving uncovered points by snapshots collected during fuzzing. Furthermore, we propose a memory abstraction to effectively reduce the computational cost of model checking and adopt footprint memory of PATHFUZZ to facilitate coverage-guided fuzzing.

We finally remark that concolic execution [8], [53] could be used to resolve uncovered points. Concolic execution is a combination of symbolic execution [7] and concrete execution, thus can be seen as a combination of formal verification and testing. Though concolic execution circumvents the state-space explosion problem, it leads to the path explosion problem when exploring uncovered points. Inspired by our work, it is interesting to study how to synergistically integrate concolic/symbolic execution and coverage-guided fuzzing in future.

VI. CONCLUSION

In this work, we introduced the concept of snapshots and proposed BMCFuzz, the first hybrid verification approach that combines synergistically BMC and CGF in a two-way fashion. The experimental results showed that BMCFuzz not only improves the coverage on three well-known RISC-V processors, but also discovers previously unknown bugs. We implemented the first open-source tool that supports hybrid hardware verification with BMC and CGF.

For the future work, we believe that the approach proposed in this paper can be adapted to and implemented in the non-DiffTest based open-source CGF tools such as Cascade [37] and DIFUZZRTL [34]. Furthermore, we plan to improve the performance of BMCFuzz further and use it to verify the more complex processor designs, e.g. XiangShan [22].

ACKNOWLEDGMENT

This work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320101.

REFERENCES

- [1] R. Saravanan and S. M. P. Dinakarrao, "The emergence of hardware fuzzing: A critical review of its significance," *arXiv preprint arXiv:2403.12812*, 2024.
- [2] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas, "Effective theorem proving for hardware verification," in *TPCD*, 1994, pp. 203–222.
- [3] S. Huang and K. Cheng, *Formal equivalence checking and design debugging*. Springer, 2012.
- [4] S. Shen, Y. Liu, L. Zhang, F. Song, and Z. Wu, "Formal verification of RISC-V processor chisel designs," in *SETTA*, 2025, pp. 142–160.
- [5] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer, 2018.
- [6] Y. Naveh, M. Rimón, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI magazine*, vol. 28, no. 3, pp. 13–13, 2007.
- [7] N. Bruns, V. Herdt, and R. Drechsler, "Processor verification using symbolic execution: A RISC-V case-study," in *DATE*, 2023, pp. 1–6.
- [8] Y. Lyu and P. Mishra, "Scalable concolic testing of RTL models," *IEEE Transactions on Computers*, vol. 70, no. 7, pp. 979–991, 2021.
- [9] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "HyPFuzz: Formal-Assisted processor fuzzing," in *USENIX Security*, 2023, pp. 1361–1378.
- [10] N. F. Dipu, M. M. Hossain, K. Z. Azar, F. Farahmandi, and M. Tehranipoor, "Formalfuzzer: Formal verification assisted fuzz testing for soc vulnerability detection," in *ASP-DAC*, 2024, p. 355–361.
- [11] "Nutshell RISC-V CPU," 2019. [Online]. Available: <https://github.com/OSCPU/NutShell>
- [12] "Rocket chip RISC-V CPU generator," 2023. [Online]. Available: <https://github.com/chipsalliance/rocket-chip>
- [13] "RISC-V Boom: The Berkeley out-of-order RISC-V processor," 2023. [Online]. Available: <https://github.com/riscv-boom/riscv-boom>
- [14] J. Bachrach, H. Vo, B. C. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzyniek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," in *DAC*, 2012, pp. 1216–1225.
- [15] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi, "Processorfuzz: Processor fuzzing with control and status registers guidance," in *HOST*, 2023, pp. 1–12.
- [16] A. Waterman, K. Asanović, and J. Hauser, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 20211203," 2021.
- [17] Y. Xu, Z. Yu, K. Wang, H. Wang, J. Lin, Y. Jin, L. Zhang, Z. Zhang, D. Tang, S. Wang *et al.*, "Functional verification for agile processor development: A case for workflow integration," *Journal of Computer Science and Technology*, vol. 38, no. 4, pp. 737–753, 2023.
- [18] K. Laeuffer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of RTL on fpgas," in *ICCAD*, 2018, pp. 1–8.
- [19] K. Laeuffer, V. Iyer, D. Biancolin, J. Bachrach, B. Nikolić, and K. Sen, "Simulator independent coverage for RTL hardware languages," in *ASPLOS*, 2023, pp. 606–615.
- [20] Y. Xu, S. Wang, D. Tang, N. Sun, and Y. Bao, "Pathfuzz: Broadening fuzzing horizons with footprint memory for CPUs," in *DAC*, 2024.
- [21] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *CCS*, 2022, pp. 1051–1065.
- [22] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen, L. Gou, Y. Jin, Q. Li, X. Li, Z. Li *et al.*, "Towards developing high performance RISC-V processors using agile methodology," in *MICRO*, 2022, pp. 1178–1199.
- [23] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.
- [24] "Spike, a RISC-V isa simulator," 2023. [Online]. Available: <https://github.com/riscv-software-src/riscv-isa-sim>
- [25] "DiffTest: a modern co-simulation framework for risc-v processors." 2019. [Online]. Available: <https://github.com/OpenXiangShan/diffTest>
- [26] A. Biere, "Tutorial on world-level model checking," in *FMCAD*, 2020.
- [27] C. Wolf *et al.*, "Symbiosys," 2022. [Online]. Available: <https://symbiosys.readthedocs.io/>
- [28] Y. Su, Q. Yang, and Y. Ci, "Predicting lemmas in generalization of IC3," in *DAC*, 2024, pp. 208:1–208:6.
- [29] "Hardware Model Checking Competition 2024," 2024. [Online]. Available: <https://hwmcc.github.io/2024/hwmcc24slides.pdf>
- [30] A. Niemetz and M. Preiner, "Bitwuzla," in *International Conference on Computer Aided Verification*. Springer, 2023, pp. 3–17.
- [31] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "Cadical 2.0," in *CAV*, 2024, pp. 133–152.
- [32] chipsalliance, "Random instruction generator for RISC-V processor verification," 2018. [Online]. Available: <https://github.com/chipsalliance/riscv-dv/>
- [33] RISC-V, "riscv-tests," Jan. 2015. [Online]. Available: <https://github.com/riscv-software-src/riscv-tests>
- [34] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find CPU bugs," in *S&P. IEEE*, 2021, pp. 1286–1303.
- [35] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," in *USENIX Security*, 2022, pp. 3219–3236.
- [36] J. Wang, B. Cui, R. Dong, and R. Zhai, "Feedbackfuzz: Fuzzing processors via intricate program generation with feedback engine," in *ICASSP*, 2025, pp. 1–5.
- [37] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU fuzzing via intricate program generation," in *USENIX Security*, 2024, pp. 5341–5358.
- [38] M. Rostami, M. Chilese, S. Zeitouni, R. Kande, J. Rajendran, and A.-R. Sadeghi, "Beyond random inputs: A novel ML-based hardware fuzzing," in *DATE*, 2024, pp. 1–6.
- [39] N. Bruns, V. Herdt, D. Große, and R. Drechsler, "Efficient cross-level processor verification using coverage-guided fuzzing," in *ACM Great Lakes Symposium on VLSI*, 2022, pp. 97–103.
- [40] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *USENIX Security*, 2023, pp. 1307–1324.
- [41] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *USENIX Security*, 2022, pp. 3237–3254.
- [42] "Questa Verification Environment," 2023. [Online]. Available: <https://eda.sw.siemens.com/en-US/ic/questa>
- [43] "VC Formal," 2023. [Online]. Available: <https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html>
- [44] "Jasper RTL Apps," 2023. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html
- [45] OneSpin, "OneSpin 360 DV RISC-V Verification App," 2020. [Online]. Available: <https://www.onespin.com/solutions/risc-v>
- [46] Axiomise, "FormalISA: RISC-V formal verification," 2023. [Online]. Available: <https://www.axiomise.com/riscv-formal-app/>
- [47] YosysHQ, "RISC-V Formal Verification Framework," Nov. 2016. [Online]. Available: <https://github.com/YosysHQ/riscv-formal>
- [48] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, "A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors," in *DAC*, 2020, pp. 1–6.
- [49] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri, "Formal security verification of third party intellectual property cores for information leakage," in *VLSI*, 2016, pp. 547–552.
- [50] R. Metta, R. K. Medicherla, and S. Chakraborty, "Bmc+fuzz: Efficient and effective test generation," in *DATE*, 2022, pp. 1419–1424.
- [51] P. K. Nalla, R. K. Gajavelly, J. Baumgartner, H. Mony, R. Kanzelman, and A. Ivrii, "The art of semi-formal bug hunting," in *ICCAD*, 2016, pp. 1–8.
- [52] M. Ganai, P. Yalagandula, A. Aziz, A. Kuehlmann, and V. Singhal, "Siva: A system for coverage-directed state space search," in *JETTA*, 2001, pp. 11–27.
- [53] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on RTL models," in *DATE*, 2018, pp. 1538–1543.