# 应对硬件设计符号执行中的路径爆炸问题

Kaki Ryan美国北卡罗来纳大学教堂山分校，美国北卡罗来纳州教堂山

Cynthia Sturton 美国北卡罗来纳大学教堂山分校，美国北卡罗来纳州教堂山

arXiv:2304.05445v1 [cs.CR] 11 Apr 2023

**摘要—符号执行是用于硬件设计验证的强大工具，但会遭受路径爆炸问题。我们提出了一种新的方法——分段组合，该方法利用硬件的模块化结构，将路径探索工作转移给 SMT 求解器。我们介绍了一个实现该技术的符号执行引擎。该引擎直接对寄存器传输级（RTL）Verilog 设计进行操作，无需转换为网表或进行软件仿真。在评估中，分段组合将探索的路径数量减少了一个数量级，并将运行时缩短了 97%。利用文献中的 84 个属性，我们在 5 个开源设计中发现了断言违例，其中包括一个片上系统和一个 CPU。**

**索引词——验证，形式化方法，硬件，安全性**

## I. 引言

硬件设计的验证是确保设计在硬件生命周期早期具备正确性和安全性的关键工作。目前的最佳实践包括基于断言的验证（ABV） [1]，其底层验证手段是基于仿真的测试；以及形式化验证技术，这一总称涵盖了许多旨在证明设计某项给定属性的技术。近年来，一种技术尤其受到关注，特别是在安全验证应用中，即符号执行 [2] [3] [4]。

符号执行通过将输入值替换为符号来推广测试，其中每个符号代表输入参数所有可能取值的集合。符号执行引擎依据程序语言的语义执行符号执行，并对语义进行扩展以支持符号。随着执行的进行，符号被用来代替字面值。对设计执行一个时钟周期的符号执行，结果是一棵由多条路径组成的树，每条路径都对应一个唯一的路径条件，用于描述沿该路径经过的分支所满足的条件。如果发现任何路径违反给定的断言，那么相应的路径条件就能精确描述驱动（具体）执行沿同一路径运行的输入；满足该路径条件的具体取值就是该断言的反例。

遗憾的是，符号执行受到路径爆炸问题的困扰——设计中的每条路径都会被单独探索，而路径数量会随着设计中分支点（或控制流语句）数量的增加呈指数增长。以往的工作试图通过将符号执行与模型检测相结合 [5],具体执行轨迹 [4], 或者将其使用限制在小型设计中 [6]来避免路径爆炸问题。

我们引入分段组合技术，利用硬件设计的结构将问题转化为可满足性模理论（SMT）求解领域，使符号探索的路径数量仅随任一 always 块中的分支点数量呈指数增长，并随设计中的 always 块数量呈线性增长。通过这种方式，我们既能受益于 SMT 领域的最新进展，又能保留在寄存器传输级（RTL）获取单条路径信息的实用性。

符号执行与符号仿真密切相关 [7][5] [8]。在这两种方法中，具体输入值都会被替换为表示任意可能值的符号值，并跟踪符号值如何在设计中传播。然而，两者存在一个关键区别。在符号仿真中，分析以数据流为中心。在一次仿真运行结束时，每个信号可能具有 true、false 或布尔表达式形式的值，该布尔表达式刻画了驱动特定信号的整个电路。电路中存在控制点时，它们会在布尔表达式中表示为 ITE 语句。在符号执行中，分析以控制流为中心。在一次迭代结束时，每个信号都由一阶逻辑中的表达式刻画，该表达式描述了通过 Verilog RTL 所采取的特定路径。此外，还存在一个路径条件，用于表示执行沿着设计中的特定路径时所需满足的条件。

在发送给 SMT 求解器的查询复杂度（符号仿真）与需要探索的路径数量（符号执行）之间，需要做出权衡。通过分段组合，我们探索了设计空间中的一个新方向，将需要探索的路径数量减少到可处理的程度，同时仍使 SMT 查询足够简单，以便现代求解器能够处理。最终得到的符号执行引擎能够处理大型设计，并直接对寄存器传输级的 Verilog 进行操作。

分段组合通过识别设计的相互独立部分来发挥作用：对于每条从根到叶的路径，这些部分无需被重复探索。该算法对每个独立的 Verilog模块进行一次符号探索，而不考虑其余模块，从而生成一组符号执行树。为了重构完整的从根到叶的路径，无论是用于查找断言失败、描述信息流如何穿过设计，还是生成测试用例，我们都可以使用 SMT 查询来组合独立探索得到的路径片段。

或许令人惊讶的是，我们证明，对于一个包含 $N$ 个 always 块的设计，其中每个块至多有 $b$ 个二元分支点，对该设计进行单个时钟周期的符号执行需要对$O(2^bN)$ 条路径进行符号执行，而不是符号执行中常见的 $O(2^{bN})$ 条路径。需要探索的路径数仅随任一独立块中的分支点数量呈指数增长，并随块的数量呈线性增长。

我们应用分段组合对五个开源设计进行符号探索，其中包括片上系统和CPU 设计，以发现设计中的断言违例。使用文献中的84 个断言，我们发现，与传统符号执行技术相比，分段组合在不损失有效性的情况下，平均可将运行时缩短 97%。

本文的贡献如下：(1) 引入并定义分段组合，这是一种利用硬件设计模块化特性的技术，旨在解决符号执行中的路径爆炸问题。(2) 使用分段组合为 Verilog 寄存器传输级设计并实现一个符号执行引擎，

合。(3) 在五种开源片上系统和 CPU 设计上评估分段组合及我们的实现。

## II. 背景

我们回顾了符号执行和SMT 求解的一般技术，并介绍了 Verilog 硬件描述语言的关键方面。

### A. 符号执行

在符号执行 [9], 中，具体字面值会被替换为符号值：输入值会被设为符号值，而符号执行引擎会“执行”设计，跟踪代码每一行的当前执行状态。执行状态有两个主要组成部分：符号存储 $\sigma$ 和路径条件 $\pi$:

1) $\sigma$，符号存储维护程序变量与符号表达式之间的映射。2) $\pi$，路径条件是关于符号表达式的布尔公式，用于描述沿当前路径所采取分支满足的条件。路径条件始终初始化为 True。

当符号执行引擎执行每一行代码时，会在字面值出现的地方用符号代替，并更新符号状态。当遇到条件为 $b$ 的分支语句时，会检查路径条件 $\pi$。如果 $\pi \to b$，则执行 then 分支。如果 $\pi \to \neg b$，则执行 else 分支（如果存在）。如果这两个蕴含关系都不成立，则必须依次探索两个分支，将当前路径分叉为两条独立路径进行探索。要探索 then 分支之后的路径，会更新路径条件 $\pi$： $\pi := \pi \wedge b$。要探索 else 分支之后的路径，则更新路径条件： $\pi := \pi \wedge \neg b$。在每个分支点，待探索的路径数量都会翻倍。这就是路径爆炸问题，通常会使用启发式方法或合并策略 [5], [10], [37] 来引导探索，以最大限度地提高覆盖率或探索深度。

对硬件设计的完整探索相当于执行一个时钟周期。硬件持续运行，潜在的安全漏洞可能要经过多个时钟周期后，才会从初始状态显现出来。这就需要对设计进行跨多个时钟周期的符号执行，从而加剧路径爆炸问题。

重要的是，我们的符号执行引擎直接对 Verilog RTL 进行操作，不会将其转换为 C 或编译成网表。这样一来，发现的任何断言违例都更易于人工理解——执行路径以及输入信号所受的约束，都可以直接通过 RTL 设计进行追踪。我们的符号执行引擎具有周期精确性。我们假设不存在组合锁存器和异步复位，并且 always 块均以输入时钟为条件。这些假设与该领域的既有工作一致 [5]。

### B. 约束求解

布尔可满足性（SAT）求解器是判定过程，它接收命题公式，并判断是否存在一组布尔值赋值，使该公式求值为真。模理论可满足性（SMT）求解器通过支持表达能力更强、能够涵盖比简单原子命题更多功能的理论，扩展了 SAT 求解。例如，支持数组运算和线性算术。SMT 求解器已变得相当强大，能够处理含有数百个变量的表达式。然而，SMT 求解仍然是一个 NP 完全问题。

SMT 求解器在符号执行中至关重要：引擎推进时，它们用于检查路径的可行性；对于找到的路径，它们还用于为符号变量生成赋值，例如生成测试用例。最广泛使用的求解器包括 STP [11] （用于 KLEE [13]）和 Z3 [12] （用于 Mayhem [14] 和 angr [15]）。

### C. Verilog

Verilog 是一种硬件描述语言，也是开发实际计算机系统的行业标准。Verilog 中设计的基本单元是模块。模块通常包含其他模块，因此设计具有层次结构。模块通过连接线和寄存器，将一个模块的输出信号连接到另一个模块的输入信号，从而组合多个子模块。Verilog 提供了多种结构，使数据流动方式不同于仅按顺序执行的软件程序。例如，多个并行组合的模块确实会并行执行。在模块内部，使用 always 块定义一组仅在特定条件下发生的事件。例如，仅在时钟上升沿或下降沿执行的赋值语句。顺序 always 块中的语句均并行执行，两个不同的 always 块也会并行运行。

## III. 分段组合

符号执行有两项主要开销：模拟执行并为每行代码更新符号存储所需的时间，以及在路径中的每个分支点判断两个分支是否都可行所需的时间。这两部分开销都很大，但近年来 SMT 求解方面的进展已降低了判断分支可行性的开销，而符号执行一条指令的开销则相对稳定。

在传统的符号执行中，每行代码都可能被访问多次，每次对应所探索的一条路径。我们的方法是将设计积极分解为独立代码块，对每个代码块进行一次符号探索，然后使用SMT 求解器组合各个代码块的路径条件和符号存储。这一策略得益于硬件设计固有的模块化特性，并使我们能够利用现代 SMT 求解器相对于符号执行代码行所需成本的较高速度。

结果是一个完整的、由穿过该设计的路径构成的（逻辑）树。虽然完整树中的路径数量相对于设计中的分支数呈指数增长，但符号执行引擎探索的路径数量仅相对于任意单个独立块中的分支数呈指数增长，并相对于块数呈多项式增长。

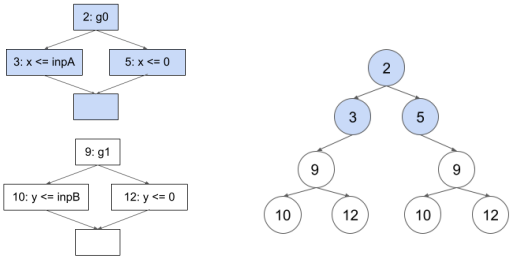
### A. 动机示例

图 1 展示了一段代码，其中包含两个 always 块，且第 2 行和第 9 行有分支点。图 2a 给出了 always 块顺序任意排列时对应的控制流图，图 2b 则展示了穿过设计的路径树。采用传统符号执行时，会对图 2b 中四条根到叶路径分别执行符号执行。这是当前方法（例如 [3], [39]）所采用的策略：先将设计转换为 C++ 表示，再使用 KLEE 符号执行引擎。但是，以标记为 9 的节点为根的两个子树代表了重复工作。对于每个子树，符号执行引擎都在探索穿过同一代码块的路径：从第 9 行开始的第二个 if-else 块。

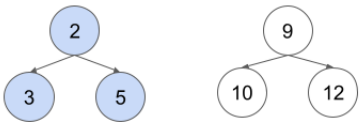
第2至5行中的分支条件和赋值与第9至12行中的分支条件和赋值相互独立。无论第一个分支（第2行）采取哪条路径，

```
1  always @ ( 上升沿 c l k ) 开始
2      if ( g0 )
3          x  <=输入信号 inpA ; / / 输入信号 inpA 是一个输入信号
4      e l s e
5          x  <=  0;
6  end
7
8  always @ ( 上升沿 c l k ) b e g i n
9      i f   (g1)
10         y  <= 输入信号 inpB;      // 输入信号 inpB   是一个输入信号
11     e l s e
12         y  <=  0 ;
13 end
```

图 1：包含两个分支的 Verilog 代码示例。




(a) 控制流图　　(b) 完整的路径树




(c) 在以下情况下独立探索的路径树分段组合

图 2：分段组合

从第二个分支点（第 9 行）开始进行符号执行，会生成相同的子树。第二个条件的路径可行性相同，对符号状态的更新也相同。例如，假设输入信号 inpA 和输入信号 inpB 分别以符号值 $\alpha$ 和 $\beta$ 初始化，并且假设在代码的这一位置，门控信号 g0 和 g1 的符号值分别为 $\gamma_0$ 和 $\gamma_1$。对于两个分支都被执行的路径（符号执行树中的节点 $\langle 2,3,9,10\rangle$），路径结束时的符号存储为 $\sigma = \{x := \alpha, y := \beta\}$，路径条件为 $\pi = \gamma_0 \wedge \gamma_1$。对于第一个分支未被执行、但第二个分支被执行的路径（$\langle 2,5,9,10\rangle$），路径结束时的符号存储为 $\sigma = \{x := 0, y := \beta\}$，路径条件为 $\pi = \neg\gamma_0 \wedge \gamma_1$。在这两条路径中，对 $y$ 的更新相同，尽管对 $x$ 的更新不同。

### *B.* 分段组合

通过分段组合，符号执行引擎会分别探索 RTL 中的独立代码块，生成彼此独立的路径片段树。在上述示例中，分段组合】【。分段组合产生了图 2c 中所示的两棵树。符号执行引擎现在仅探索第二个 if-else 块一次。继续我们的示例，分段组合将分别探索两个 always块，生成以下四个路径片段（标记为 $a$ 至 $d$），以及相关的路径条件和（部分）符号存储：

$$\begin{aligned}
\langle 2,3\rangle_a: &\quad \sigma_a = \{x := \alpha\}, \pi_a = \gamma_0 \\
\langle 2,5\rangle_b: &\quad \sigma_b = \{x := 0\}, \pi_b = \neg\gamma_0 \\
\langle 9,10\rangle_c: &\quad \sigma_c = \{y := \beta\}, \pi_c = \gamma_1 \\
\langle 9,12\rangle_d: &\quad \sigma_d = \{y := 0\}, \pi_d = \neg\gamma_1
\end{aligned}$$

为了找到贯穿设计的完整路径并成功发现断言违例，借助 SMT 求解器将所有可实现的路径片段组合起来。比如，为了实现路径 $\langle 2,5,9,10\rangle$，符号执行引擎会查询 SMT 求解器，以确定两个路径片段 $\langle 2,5\rangle$ 和 $\langle 9,10\rangle$ 是否可以连接：isSAT($x = 0 \wedge \neg\gamma_0 \wedge y = \beta \wedge \gamma_1$)。在这个简单示例中，路径片段的四种组合都是可能的，但一般情况下并不总是如此。例如，如果门控信号 g0 和 g1 都取自同一个输入，因此始终被约束为具有相同的值，那么 SMT 求解器会发现路径 $\langle 2,5,9,10\rangle$ 和 $\langle 2,3,9,12\rangle$ 是不可实现的。

分段组合的一个重要特性是其可靠性：所有可实现的组合路径，都对应于可通过完整设计重放的路径。SMT 求解器返回的满足解可以作为输入，在起始复位状态下输入设计，并使执行（无论是在仿真中还是在 FPGA 上运行）遵循相应的路径。

### *C.* 与回溯和缓存的比较

分段组合与回溯和缓存有一些相似之处；这两种技术常用于软件符号执行引擎（例如 KLEE [13], Angr [15]）。不过，两者之间存在关键差异。回溯通过在路径上的每个位置保存状态，并允许具有共同前缀的两条路径复用已保存的状态，来减少重复工作。例如，如果已探索路径 $\langle 2,3,9,10\rangle$，那么当引擎探索路径 $\langle 2,3,9,12\rangle$时，回溯可以让引擎复用位置 9 处保存的状态，并从那里继续探索。回溯避免了针对 $\langle 9,10\rangle$ 和 $\langle 9,12\rangle$中的每一条路径都重新探索路径 $\langle 2,3\rangle$；分段组合也能避免这种重复探索。然而，使用回溯时，路径$\langle 9,10\rangle$和 $\langle 9,12\rangle$ 仍会被重新探索，以生成以此前缀 $\langle 2,5\rangle$开头的路径；分段组合则能避免这种重复探索。

通过重用先前查询的结果，缓存查询可以减少在 SMT 求解器中花费的时间。缓存查询是一种与分段组合正交的技术。结合使用这两种技术可以进一步缩短运行时。

## IV. 采用分段组合的符号执行引擎

在具体实现上，符号执行引擎通过将设计分解为多个分区来实现分段组合：一个分区用于包含设计中的所有组合逻辑，一个分区用于包含所有寄存器声明，以及 $N$ 个分区，每个 always 块对应一个，用于处理设计中的时序逻辑。除下文 IV-A 节将讨论的组合逻辑分区外，每个分区均在每个时钟周期内进行一次完整的符号探索。

$N$ 个时序 always 块分区分别独立于其他 always 块进行探索，探索会生成一组路径片段。对完整设计进行完整探索后，会生成$N$ 组结果，每个 always 块对应一组。设计中从根节点到叶节点的完整符号执行路径集合，是通过对这$N$ 组路径片段取笛卡尔积形成的。SMT 求解器用于确保仅保留合理的组合，即那些对应于设计中真实路径的组合。

### *A.* 组合逻辑

符号执行引擎会检查是否存在任何组合锁存器，如果发现，则会退出并报告错误。否则，该工具会对分区中的每条组合逻辑语句执行符号执行，然后开始对穿过每个always块的控制流路径执行符号执行。在执行每个块时，引擎会跟踪每个信号的脏位；当信号被更新时，其脏位会被设为1。路径完成后，组合逻辑分区中所有右侧涉及脏信号的赋值语句都会按顺序重新求值。在重新求值期间，引擎会继续跟踪信号变为脏信号的时刻。此外，在此阶段求值时，使用三元运算符的条件连续赋值会被标记为 ITE 语句，而不是分支点。

### *B.* 时序逻辑

每个时序always块都会被独立探索。如果这些 always 块确实相互独立——无论其他块所采取的路径如何，一个块中的各条路径的路径条件和符号状态都相同——这种方法就是可靠的。下面我们将更详细地讨论独立性问题。考虑两个时序 always 块，$B_0$ 和$B_1$, , 两个块都在输入时钟信号的上升时钟边沿触发。（使用哪个时钟边沿并不重要，但假设两个块在同一个时钟信号的同一个边沿触发。）

**独立性** 在最简单的情况下，出现在 $B_0$ 中的变量均不会出现在 $B_1$中。这两个块相互独立，并且在单个时钟周期内，一个块的执行不会影响另一个块的执行。（请记住，两个时序 always 块可以被视为并行执行。）这两个块可以分别探索，其路径可以按任意顺序组合。不过，这种情况很少见，因为输入复位信号通常会出现在所有或大多数块中。

**读-读依赖** 在下一种情况下，同一变量可能同时出现在$B_0$ 和$B_1$中的分支条件或赋值右侧。这两个块仍可分别探索，并且可以按任意顺序组合其路径。然而，某些路径组合可能不可行，因为若两个块的分支条件中都出现变量，例如分别为$b_0$ 和$b_1$, , 那么当$b_0 \wedge b_1$ 不可满足时，这些变量会阻止将$B_0$ 中$b_0$ 成立的路径与$B_1$ 中$b_1$ 成立的路径组合起来。

**读写依赖** 在下面这种情况下，变量可能出现在分支条件中，或出现在$B_0$ 中赋值语句的右侧，并出现在$B_1$中赋值语句的左侧。使用非阻塞赋值时，对$B_1$ 中变量的更新会在下一个时钟周期生效，而$B_0$ 中的读取和条件分支使用的是上一时钟周期设定的变量值。符号执行引擎会保留正确的值，因此不会发生冲突。可以分别探索这两个块，并按任意顺序组合它们的路径。我们的工具不支持在时序 always 块中使用阻塞赋值。

**写写依赖** 在最后一种情况下，变量同时出现在两个 always 块中赋值语句的左侧。这违反了 Verilog 设计的最佳实践。符号执行引擎会检查是否存在写写依赖；如果发现任何此类情况，就会报错退出。

### *C.* 多个时钟周期

对硬件设计执行一个时钟周期的符号执行会生成一棵树，其中根节点表示处于复位状态或初始状态的设计，而每个每个叶节点对应于时钟周期边界处的一个“下一状态”。要探索设计两个时钟周期，就意味着再次探索该设计，即针对在时钟周期0 期间构建的树的每个叶节点分别进行一次探索。这次探索不会从复位状态开始，而是从前一周期给定叶节点所指示的状态开始。在每次单个时钟周期的探索结束时，都会进行一个合并过程：所有输入信号和组合连线都会获得新的符号，而有状态元素则沿用上一周期的表达式。

### *D.* 进一步优化

#### *1)* 重复子模块

当这些模块是同一模块的重复实例化时，总体搜索空间有缩减的余地。该优化的主要思想与分段组合类似，即对于每条路径，我们只探索每个子模块一次。这样，与其针对每个重复模块实例再次进行探索，我们可以使用 SMT 查询，将给定根到叶路径的符号存储和路径条件合并起来。

#### *2)* 影响锥分析

此优化在块级剪枝探索空间。符号执行引擎会读取断言中提供的表达式，对断言中的信号进行依赖分析，然后完成 AST 遍历，以确定哪些块会读取或写入感兴趣的信号或其依赖项。完成这次初始遍历后，引擎将只探索涉及感兴趣的信号或其依赖项的块。

## V. 实现

我们的符号执行引擎基于Python 3.8 构建，并根据 IEEE 1364-2005 标准实现 Verilog语义。我们使用 pyVerilog 库构建 Verilog 抽象语法树，并使用 networkX 管理图搜索和遍历。我们使用Z3 Python API进行SMT 求解。该符号执行引擎接收以 Verilog 编写的设计（包括根据 SystemVerilog 1800-2017 标准编写的断言），并输出可重放的反例。

## VI. 评估

我们在五种开源设计上评估了实现，其中包括 CPU 和片上系统设计，以研究其作为硬件设计验证平台的可行性。我们的评估考察以下问题：1）分段组合在多大程度上能够缓解路径爆炸问题？2）分段组合和第 IV-D 节所述的优化对性能有何影响？3）符号执行引擎能否针对存在缺陷和漏洞的设计生成带有可重放反例的断言违例?

*A.* 数据集与实验设置

我们从多个来源收集了五个设计和 84 条安全关键断言。前三个设计及其相关断言来自 TrustHub [16][17] 提供的安全属性/规则数据库：适用于 Motorola MC68HC11 系列 CPU 的增强版串行外设接口；openMSP430，一款可综合的 16 位微控制器内核，与 Texas Instruments 的 MSP430 微控制器系列兼容；以及 CrypTech 真随机数生成器（TRNG）。对于这些设计，数据库分别包含 9、2 和 2 条安全属性。

第四个设计是最近一次 Hack@DAC 竞赛中使用的 PULPissimo片上系统 [18]。该设计存在漏洞；其中一些漏洞由竞赛组织者手动植入，另一些则是设计本身固有的 [19]。我们利用这些属性的英文描述，以及寄存器传输级-ConTest论文 [4] 中对测试用例生成过程的演示，开发了 26 条断言以供我们的工具使用。

第五个设计是 OR1200 处理器核。我们从两篇先前的论文中收集了 30 个安全关键漏洞：SPECS [20] 和 SCIFinder [21]；并从 SPECS [20], Security Checkers [22], SCIFinder [21], 和 Transys [23]中收集了 70 条安全断言。

实验在一台配备英特尔 Xeon E5-2620V312 核 CPU（2.40GHz，双路服务器）且有 62G 可用 RAM 的机器上进行。

*B.* 路径爆炸问题的缓解

我们评估了分段组合缓解路径爆炸问题的效果。设计中现存的路径数量不会改变，但借助分段组合，符号执行引擎为实现一条完整路径所需完成的工作量要小得多。对于每个设计，我们比较了在有无分段组合的情况下，为找到断言违规而访问的平均代码行数和分支点数量。表I 给出了结果。分段组合将访问的代码行数（即减少对同一代码行的冗余访问）降低了 92%–99%，并将访问的分支点数量降低了 64%–99%。

为了获得更完整的图景，我们对一个设计中的所有路径进行了符号探索。我们使用了 MC68HC11SPI，该设计足够小，因此无需分段组合的符号执行即可完成探索。该设计包含 13 个 always 块，以及贯穿该设计的 1459条路径。这些结果报告于表 II 中。没有分段组合时，探索了超过 9 万行代码和超过 7 千个分支点。采用分段组合后，符号执行引擎只需探索约占这 9万行代码 7% 的代码，以及约占这 7 千个分支点 4% 的分支点。

*C.* 优化的影响

图 3 展示了分段组合对每个设计的 SMT 查询平均次数以及 SMT 求解器平均耗时的影响。有人可能会担心，减少冗余探索所带来的收益会以 SMT 求解器工作量激增为代价。然而，实际情况恰恰相反。减少探索的路径数，也会减少对求解器的查询总次数；请记住，在探索过程中，每个分支点都会查询求解器。启用分段组合后，SMT 查询次数平均减少了 18%，求解耗时减少了 21%。

在表III中，我们测量四种情况下的运行时：基线，未启用任何优化；分段组合，启用分段组合；重复模块，仅探索一次重复模块；以及影响锥分析，在探索之前完成影响锥分析。每种情况均是累积的，例如，在重复模块情形下，并行优化也会启用。对于每个设计，我们在考察各个断言时取平均值。除了最小的设计外，基线均无法在 30 分钟内可靠地完成探索，因此我们在此时停止搜索。表 III 展示了结果。对于每种情况，我们给出以秒为单位的绝对运行时，以及相较于前一种情况运行时的降幅。总体而言，这些优化使引擎的运行时减少了 95-99%。

*D.* 查找断言违规

为了评估引擎查找断言违例的能力，我们进行了一系列实验；对于每个设计，我们都掌握其违规数量（最小值）的真实值信息。在我们的数据集中，有两个设计具有真实值信息：Hack@DAC 片上系统（31 个漏洞）和 OR1200 处理器核（30 个漏洞）。

在这些实验中，符号执行从复位状态开始，将所有输入信号设为符号化，并持续执行，直到发现断言违规。表 IV 总结了结果。

该引擎发现了 Hack@DAC 片上系统中的31个漏洞中的24 个。相比之下，先前的工作报告称，使用混合具体符号执行发现了 31 个漏洞中的 14 个 [4]，而 Hack@DAC 的组织者报告称，使用商业工具Cadence SPV 和Cadence FPV 分别发现了 6 个和 15 个漏洞 [18]。

该引擎在 OR1200 中发现了 30 个漏洞中的 27 个。在漏掉的 3 个漏洞中，有一个没有相应属性覆盖。第二个漏洞与正确执行的指令有关，攻击者可以利用内核提前退出实施 ROP 漏洞利用。第三个漏洞与控制流有关，会错误地设置链接寄存器，而处理器从函数调用返回时会用到该寄存器。

我们的引擎生成的所有反例，都已使用 Vivado 从复位状态开始在仿真中成功重放。

*E.* 与当前先进水平的比较

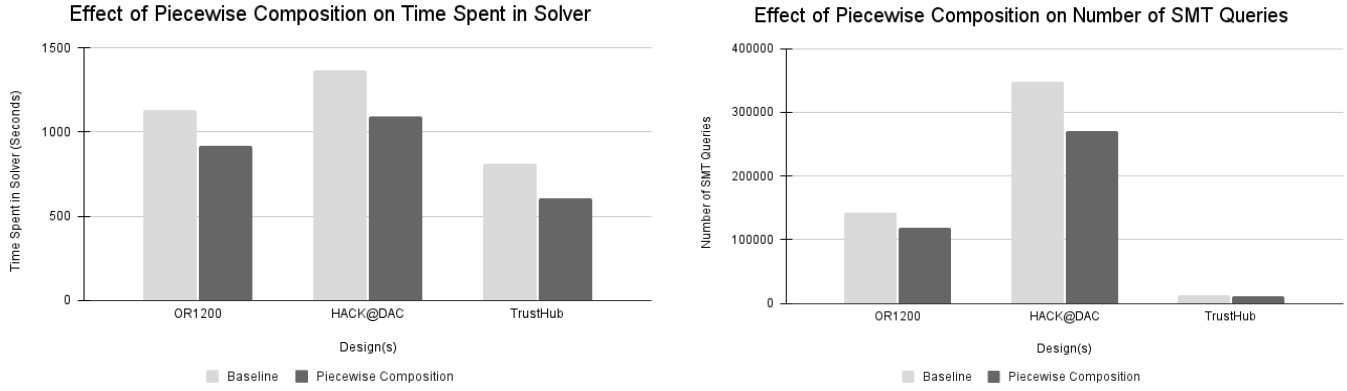
Coppelia 是一种工具，首先将 Verilog 转换为 C，然后采用传统的符号执行，不过其搜索策略针对硬件设计进行了优化 [3]。作者报告称，在他们的实验中，大多数（62%）漏洞利用都在 15 分钟内生成，另有一些在 2–4 小时内找到。相比之下，采用分段组合的符号执行在可比的设计（包括 OR1200）中，不到 2 分钟便能找到相同的漏洞利用。RTLConTest [4], 的作者报告称，他们的混合具体符号执行引擎在 PULpissimo 片上系统上运行约需 1 小时 40 分钟才能完成；该片上系统是 HACK@DAC 2018 设计的一个修改版本，我们在评估中使用了该设计。完成混合具体符号执行并生成测试后，平均只需 10 秒即可生成一个反例。针对相同漏洞的安全属性，我们的工具通过完整的端到端符号执行工作流生成反例，平均耗时 122 秒。

## VII. 相关工作

**模型检测** 当前的硬件验证实践涉及基于仿真的测试与静态分析技术（如模型检测）的结合。模型检测能够正式验证程序是否满足一组以时序逻辑公式形式给出的规范 [24]。通常的验证过程是对设计空间进行穷举搜索。实践中采用有界模型检测，验证过程会探索设计在最多 k 个时间步内的行为 [25]。符号轨迹评估是一种基于格的模型检测技术，由英特尔开发并使用 [26], , 近期的研究已将其进一步推进到字级层面 [27]。STE 的目标是通过有界长度轨迹，对时序系统的属性进行形式化验证

| 设计 | 基线 | | 分段组合 | | 下降百分比 | |
|---|---|---|---|---|---|---|
| | LoC 探索了 | 分支点 探索了 | LoC 已探讨 | 分支点 已探讨 | LoC 探索过 | 分支点 已探索 |
| OR1200 | 54018 | 7803 | 881 | 45 | 98% | 99% |
| Hack@DAC | 493032 | 15093 | 3525 | 276 | 99% | 98% |
| MC68HC11 SPI | 2093 | 158 | 174 | 57 | 92% | 64% |
| openMSP430 | 15293 | 377 | 489 | 68 | 97% | 82% |
| CrypTech 真随机数生成器 | 8930 | 421 | 336 | 91 | 96% | 78% |

表 I：分段组合对路径爆炸的平均影响




（a）求解器时间

（b）SMT 查询

图 3：优化对求解器时间和 SMT 查询次数的影响

| 配置 | LoC 已探索 | 分支点 已探索 | 路径 已完成 |
|---|---|---|---|
| 基线 | 90706 | 7380 | 1459 |
| 分段组合 | 6783 | 323 | 1459 |

表 II：MC68HC11 SPI 设计的完整探索

并采用一种经修改的三值符号仿真形式 [28]。SymbiYosys [8]是构建于 Yosys 之上的模型检测引擎，作为 Yosys [29] 现有性质检查框架的扩展。

**符号执行与混合具体符号执行**

符号执行及相关技术在寄存器传输级设计中的应用日益受到关注。RTLConTest [4] 和 Witharana 等人 [30] 的工作，都是为硬件设计安全验证而开发的混合具体符号测试引擎。Coppelia [3] 是一种面向硬件的反向符号执行引擎，它基于 KLEE 构建，用于处理转换为 C 的寄存器传输级设计++。EISec [39] 同样使用 KLEE，但用于处理转换为 C 的网表++。这些工作都展示了符号执行应用于硬件设计的强大能力，但仍然难以应对路径爆炸问题。这些论文中提出的许多技术都可以与分段组合结合使用。

**模糊测试** 模糊测试也已被证明是发现片上系统和 CPU 设计中安全漏洞的一种有效技术。RFUZZ 是一种面向电路的覆盖率导向模糊测试器，它提出了一种名为多路复用器控制覆盖率 [31]的硬件专用覆盖率指标。DifuzzRTL 是一种寄存器传输级模糊测试工具，用于发现未知的安全漏洞；它基于控制寄存器而非多路复用器控制信号来衡量覆盖率，从而提高效率和可扩展性[32]。近期开发的硬件模糊测试流水线会将寄存器传输级转换为软件模型，以提高通过模糊测试发现漏洞的可扩展性 [33]。

**信息流跟踪** 在硬件环境中，信息流指不同信号之间的信息传递。信息流跟踪是一种验证方法，用于研究信息如何在硬件设计中流动，从而对安全性作出判断 [34] [35]。可以在硬件设计中加入跟踪逻辑，以捕获时序 [36] 或数据流信息。这种方法能够提供强有力的安全保证，例如，证明密钥数据泄漏到了非预期的输出信号。

## VIII. 结论

我们介绍了分段组合，这是一种用于应对符号执行中路径爆炸问题的技术。我们采用该技术实现了一个符号执行引擎，并在五种开源设计上对该引擎进行了评估。与传统符号执行相比，该引擎将冗余工作减少了 98%–99%，提升了整体性能，并成功发现了断言违规。

## IX. 致谢

本研究得到美国国家科学基金会资助（Grant No. CNS-1816637），并获得 Meta安全研究奖支持。

## 参考文献

[1] ClaudionorNunesCoelho 和 HarryD.Foster。《基于断言的验证》，第 167–204 页。Springer US，马萨诸塞州波士顿，2004年。[2] 沈立祥、牟德军、GuoCao、MaoyuanQin、Jeremy Blackstone 和 Ryan Kastner。用于硬件木马检测的基于符号执行的测试模式生成算法。《计算机与安全》，78 280，2018年 267：–。

| 设计 | 基线 | 分段式 | | 冗余 | | COI | | 总体而言 |
|---|---|---|---|---|---|---|---|---|
| | 运行时 (秒) | 运行时 (秒) | % 降低 | 运行时 (秒) | % 降低 | 运行时 (秒) | % 降低 | % dec |
| OR1200 | 超时（1800） | 52.47 | 97.08% | 37.56 | 21.31% | 25.22 | 12.56% | 98.60% |
| Hack@DAC | 超时 (1800) | 174.24 | 90.32% | 121.94 | 28.34% | 81.83 | 16.62% | 95.45% |
| MC68HC11 | 962 | 17.53 | 98.18% | 14.30 | 19.93% | 0.07 | 99.19% | 99.99% |
| openMSP430 | 超时（1800） | 37.65 | 97.91% | 23.14 | 38.55% | 0.73 | 96.83% | 99.96% |
| CrypTech 真随机数生成器 | 超时 (1800) | 14.92 | 99.17% | 12.08 | 19.15% | 0.09 | 99.19% | 99.99% |

表 III：优化对运行时间的平均影响

| 设计 | # 漏洞 | # 漏洞找到 | Avg Time (sec) | Max 时钟周期已采用 |
|---|---|---|---|---|
| Hack@DAC | 31 | 24 | 122 | 4 |
| OR1200 | 31 | 27 | 34 | 5 |

表 IV：已知漏洞


[3] Rui Zhang、Calvin Deutschbein、Peng Huang 和 Cynthia Sturton。用于验证处理器设计安全性的端到端自动化漏洞利用生成。载于国际微架构研讨会（*MICRO*）论文集。IEEE/ACM，2018年。[4] Xingyu Meng、Shamik Kundu、Arun K. Kanuparthi 和 Kanad Basu。RTL-ConTest：在 RTL 上进行合取符号测试以检测安全漏洞。*IEEE* 集成电路与系统计算机辅助设计汇刊，41(3):466–477，2022年。[5] Anish Athalye、M. Frans Kaashoek 和 Nickolai Zeldovich。使用信息保持精化验证硬件安全模块。载于*OSDI*。USENIX 协会，2022年。[6] R. Mukherjee、D. Kroening 和 T. Melham。使用软件分析器进行硬件验证。载于*2015*年 *IEEE* 计算机协会 *VLSI* 年度研讨会，第 7–12 页，2015年。[7] Voss ii。https://github.com/TeamVoss/VossII。访问日期：2022年11月21日。[8] Symbiyosys。https://github.com/YosysHQ/sby。访问日期：2022年11月21日。[9] James C. King。符号执行与程序测试。*Commun. ACM*，19(7):385ˆaC“394，1976年7月。[10] S. Krishnamoorthy、M. S. Hsiao 和 L. Lingappaān。解决程序中符号执行驱动测试生成的路径爆炸问题。载于*2010*年第*19*届 *IEEE* 亚洲测试研讨会，第 59–64 页，2010年。[11] Blayne Mayfield 和 Timothy Baird。STP。载于*ACM* 论文集，SIGSMALL ’90，第 98–105 页，美国纽约州纽约市，1990年。计算机协会。[12] Leonardo De Moura 和 Nikolaj Bjørner。Z3：一种高效的 SMT 求解器。载于系统构造与分析工具和算法国际会议，第 337–340 页。Springer，2008年。[13] Cristian Cadar、Daniel Dunbar、Dawson R Engler 等。KLEE：为复杂系统程序生成高覆盖率测试的无需辅助且自动化方法。载于*OSDI*，第 8 卷，第 209–224 页，2008年。[14] Sang Kil Cha、Thanassis Avgerinos、Alexandre Rebert 和 David Brumley。释放二进制代码中的 Mayhem。载于*2012*年 *IEEE* 安全与隐私研讨会论文集，SP ’12，第 380ˆaC“394 页，美国，2012年。IEEE 计算机协会。[15] Soomin Kim、Markus Faerevaag、Minkyu Jung、SeungIl Jung、DongYeop Oh、JongHyupr Lee 和 Sang Kil Cha。测试用于二进制分析的中间表示。载于第*32*届 *IEEE/ACM* 自动化软件工程国际会议论文集，ASE 2017，第 353–364 页。IEEE Press，2017年。[16] Nusrat Farzana、Fahim Rahman、Mark Tehranipoor 和 Farimah Farahmandi。使用性质检查进行片上系统安全验证。载于*2019*年 *IEEE* 国际测试会议（*ITC*），第 1–10 页，2019年。[17] Nusrat Farzana 、 Farimah Farahmandi 和 Mark Mohammad Tehranipoor。片上系统安全属性与规则。*IACR Cryptol. ePrint Arch.*，2021:1014，2021年。[18] Hack@DAC 2018 SoC。https://github.com/seth-lab-tamu/hackdac- 2018-soc。访问日期：2022年2月15日。

[19] Ghada Dessouky、David Gens、Patrick Haney、Garrett Persyn、Arun Kanuparthi、Hareesh Khattri、Jason M. Fung、Ahmad-Reza Sadeghi 和 Jeyavijayan Rajendran。《HardFails：深入了解可被软件利用的硬件漏洞》。载于*USENIX ’19*，第 213–230 页，加利福尼亚州圣克拉拉，2019年8月。USENIX 协会。[20] Matthew Hicks、Cynthia Sturton、Samuel T. King 和 Jonathan M. Smith。《Specs：一种保护软件免受安全关键型处理器漏洞影响的轻量级运行时机制》。载于*ASPLOS*，ASPLOS ’15，第 517–529 页，美国纽约州纽约市，2015年。ACM。[21] Rui Zhang、Natalie Stanley、Christopher Griggs、Andrew Chi 和 Cynthia Sturton。《为处理器动态验证确定安全关键属性》。载于*ASPLOS*，ASPLOS ’17，第 541–554 页，美国纽约州纽约市，2017年。ACM。[22] M. Bilzor、T. Huffmire、C. Irvine 和 T. Levin。《安全检查器：在运行时检测处理器中的恶意植入》。载于*HOST*，2011年。[23] Rui Zhang 和 Cynthia Sturton。《Transys：在硬件设计中利用通用安全属性》。载于安全与隐私研讨会（*S&P*）论文集。IEEE，2020年。[24] Edmund M. Clarke。《模型检测》。载于 S. Ramesh 和 G. Sivakumar 编，软件技术与理论计算机科学基础，第 54–56 页，柏林、海德堡，1997年。Springer Berlin Heidelberg。[25] Edmund Clarke、Armin Biere、Richard Raimi 和 Yunshan Zhu。《利用可满足性求解进行有界模型检测》。19(1):7–34，2001年。[26] Randal E. Bryant、Derek L. Beatty 和 Carl-Johan H. Seger。《通过符号三值轨迹评估进行硬件形式化验证》。载于*ACM/IEEE DAC*，1991年。[27] Supratik Chakraborty、Zurab Khasidashvili、Carl-Johan H. Seger、Rajku- mar Gajavelly、Tanmay Haldankar、Dinesh Chhatani 和 Rakesh Mistry。《用于字级验证的符号轨迹评估：理论与实现》。形式化方法与系统设计，50(2–3):317–352，2017年6月。[28] Koen Claessen 和 Jan-Willem Roorda。《符号轨迹评估简介》。载于 Marco Bernardo 和 Alessandro Cimatti 编，硬件验证的形式化方法，第 56–77 页，柏林、海德堡，2006年。Springer Berlin Heidelberg。[29] Yosys。https://github.com/YosysHQ/Yosys。访问日期：2022年11月21日。[30] Hasini Witharana、Yangdi Lyu 和 Prabhat Mishra。《为激活 RTL 模型中的安全断言而进行的定向测试生成》。*ACM*设计自动化与电子系统汇刊，26(4)，2021年1月。[31] Kevin Laeufer、Jack Koenig、Donggyu Kim、Jonathan Bachrach 和 Koushik Sen。《RFUZZ：针对 FPGA 上 RTL 的覆盖率引导模糊测试》。载于*ICAAD*，第 1–8 页，2018年。[32] Jaewon Hur、Suhwan Song、Dongup Kwon、Eunjin Baek、Jangwoo Kim 和 Byoungyoung Lee。《DifuzzRTL：通过差分模糊测试查找 CPU 漏洞》。载于第*42*届 *IEEE S&P*，第 1286–1303 页。IEEE，2021年。[33] Timothy Trippel、Kang G. Shin、Alex Chernyakhovsky、Garret Kelly、Dominic Rizzo 和 Matthew Hicks。《像软件一样对硬件进行模糊测试》。载于*USENIX ’22)*，第 3237–3254 页，马萨诸塞州波士顿，2022年8月。USENIX 协会。[34] Armaiti Ardeshiricham、Wei Hu、Joshua Marxen 和 Ryan Kastner。《用于可证明安全的硬件设计的寄存器传输级信息流跟踪》。载于*DATE*，第 1691–1696 页，2017年。[35] Wei Hu、Armaiti Ardeshiricham、Mustafa S Gobulukoglu、Xinmu Wang 和 Ryan Kastner。《用于硬件安全验证的属性特定信息流分析》。载于*ICCAD*，ICCAD ’18，美国纽约州纽约市，2018年。计算机协会。[36] Ar maiti Ardeshiricham、Wei Hu 和 Ryan Kastner。《Clepsydra：对硬件设计中的时序流进行建模》。载于*ICCAD*，第 147 154 页，2017年 –。

[37] Drew Davidson、Benjamin Moench、Thomas Ristenpart 和 Somesh Jha。《固件上的 FIE：使用符号执行在嵌入式系统中发现漏洞》。载于 *USENIX* 安全会议，第 463–478 页，华盛顿特区，2013年8月。USENIX 协会。

[38] Alfred Kolbl、James Kukula、Robert Damiano。《符号 RTL 仿真——tion In设计自动化大会, 2001年. IEEE.

[39] Farhaan Fowze、MuhtadiChoudhury、Domenic Forte，EISec：利用符号执行实现硬件知识产权的穷尽式信息流安全，发表于亚洲硬件安全研讨会，2022年。IEEE。