arXiv:2301.01321v1 [cs.CR] 3 Jan 2023

Cheesecloth：现实世界漏洞的零知识证明

Santiago Cuéllar* Bill Harris JamesParker Stuart Pernsteiner Eran TromerGalois, Inc. Galois, Inc. Galois,Inc. Galois, Inc. 哥伦比亚大学

摘要

目前，安全分析师在关键软件系统中发现漏洞时，必须面对一个棘手的两难局面：立即向公众披露漏洞可能会损害系统用户的利益；而只向软件供应商披露漏洞，则可能让供应商忽视或降低对该安全风险的重视，从而损害那些在不知情的情况下受到影响的用户的利益。

近期一项颇具前景的研究方向旨在通过零知识（ZK）协议解决这一问题：这类协议允许分析师证明自己知道程序中的某个漏洞，同时不泄露漏洞详情或可利用该漏洞的输入。原则上，这可以通过通用零知识技术实现。但在实践中，迄今为止的零知识漏洞证明在适用范围和表达能力上仍有限制，原因在于生成能够大规模建模现实世界软件的证明陈述，以及直接表述被违反的属性，都面临挑战。

本文提出 CHEESECLOTH，这是一种新颖的证明陈述编译器。它通过对公共输入上的程序进行可靠但激进的预处理，有选择地披露已执行控制片段的信息，并使用一种新颖的存储标记方案对信息泄漏进行形式化，从而证明零知识中的实际漏洞。CHEESECLOTH 的实用性通过为关键软件（先前版本）中的知名漏洞生成零知识证明得到了展示，其中包括 OpenSSL 中的 Heartbleed 信息泄漏，以及 FFmpeg 图形框架中的一个内存漏洞。

1 引言

理想情况下，处理敏感信息的程序应始终安全可靠地执行。但在可预见的未来，这一理想仍难以实现，因此一旦发现程序存在漏洞，及时、安全地提醒受影响的用户就至关重要。这项要求带来了挑战：要令人信服地披露漏洞，似乎必须分享漏洞的详细信息（例如能够触发该漏洞的漏洞利用程序），从而在短期内使用户面临更大的风险。

一种颇具前景、既能令人信服地披露漏洞又能确保安全的方法，是利用零知识（ZK）证明：在这类协议中，一方（称为证明者）使另一方（称为验证者）相信某项主张有效，却不泄露有关该主张证据的任何额外信息。

自最初的基础性结果确立所有 NP 问题都存在零知识证明以来，这种对零知识证明的应用在概念上或许一直都是可行的 [26]。随着底层零知识协议的改进，以及用于编码以便捷语言编写的程序执行情况的方案问世，这种应用变得更加切实可行（始于 [10, 24]，下文将进一步讨论）。

为了证明实用软件中的零知识漏洞，仍有若干未解决的问题亟待处理。首先，证明框架必须能够扩展，以编译出针对需要更多执行步骤和空间的漏洞的证明。TinyRAM [10–12] 足够灵活，可以验证应用程序的执行，但成本高昂，部分原因在于它每一步都会模拟所建模中央处理器的指令集架构中的每条指令。Pantry [17] 和 Buffet [52], 的性能都优于 TinyRAM，但这两个框架都要求将循环展开至一个公开上限：公开这些上限会泄露有关底层漏洞的信息。

第二个尚待解决的问题，是如何将语句从易于理解的形式高效编译出来。一种直接的思路是，在动态安全监控器下执行程序，以检查那些已有充分研究的安全属性，例如 Valgrind [41] 中实现的属性；然而，直接对额外的监控进行编码会带来难以承受的巨大开销。在零知识 [27] 中验证底层漏洞利用的方法，依赖于能否将易于直接理解的属性高效编译为控制位置可达性语句。

*作者按字母顺序排列。

为了解决这些问题，我们提出 CHEESECLOTH，这是一种优化型零知识证明语句生成器，能够高效编码实用软件中的漏洞。CHEESECLOTH 设计的主要贡献包括：

1. 对用于验证程序执行的零知识证明语句进行优化，利用程序结构但不泄露有关执行的额外信息。具体而言，公共程序计数器片段通过带有公开程序计数器的片段构建执行轨迹，从而能够进行激进的常量折叠，而不会泄露整体执行轨迹的信息。同样，对于可公开确定为很少执行的指令，我们采用稀疏支持（即不能在每一步执行），从而缩小证明语句。

2. 为实际软件中常见的内存错误提供新颖且高效的零知识编码，具体包括越界访问、释放后使用、重复释放和未初始化访问。以往的相关工作主要侧重于证明对有效执行的知晓，而不证明漏洞的存在[10,11]，或者采用效率较低的内存模型对漏洞证明进行编码[29]。

3. 一种新颖且高效的语句编码，用于证明程序总会泄露数据（将漏洞利用程序作为秘密输入时）。我们的方案能够证明程序属性，这些属性与现有的程序监控器和类型系统所证明的属性相关，但有着关键区别；后者证明程序可能会泄露数据[20,39],，也可选择采用零知识证明 [21]。

我们在 CHEESECLOTH 中实现了这些优化和编码。CHEESECLOTH 是一套完整的编译工具链，可将现实世界程序中的漏洞编码为高效的零知识证明。这套工具链扩展了以往基于 TinyRAM 的方法，包含对一种新型 TinyRAM 扩展（名为 MicroRAM）的完整定义，以及一个可将 LLVM 中间语言编译为 MicroRAM 的编译器，从而能够为以 C、C++ 或 Rust 编写的程序中的漏洞生成证明。

我们通过在零知识中证明实际系统软件中存在三个漏洞，评估了我们的实现。具体而言，我们证明 GRIT 和 FFmpeg [2] 图形处理库的早期版本包含缓冲区溢出漏洞，并证明 OpenSSL 加密工具包 [3] 容易受到臭名昭著的 Heartbleed 漏洞 [5]的攻击。CHEESECLOTH 接收软件的 C/C++ 源代码以及一个标明漏洞类别的标志；它将这些内容与运行时环境（操作系统和库）的模拟相结合，并应用前述技术，推导出一条可直接通过零知识证明的陈述。随后，可将用于证明原始攻击的具体漏洞利用程序作为见证提供给零知识证明。CHEESECLOTH 实现了强大的程序分析技术；在某些情况下，将其与手动程序划分结合起来后，与更朴素的编译器相比，它能处理规模大得多的程序，

编译器。

本文其余部分安排如下：第 2 节回顾本研究所依据的背景知识。第 3 节介绍 CHEESECLOTH 编译流水线的实现细节；第 4 节介绍我们为验证程序的零知识执行所采用的关键且激进的优化；第 5 节描述我们用于高效检测内存和信息泄漏漏洞的零知识编码；第 6 节介绍我们使用 CHEESECLOTH 证明漏洞的实践经验；第 7 节将我们的方法与相关工作进行比较；第 8 节作出总结。

2 背景

本节将回顾我们工作所基于的已有研究，具体包括 程序执行的零知识（ZK）证明 （第 2.1节）、程序的信息泄露（第 2.2节）以及部分程序求值（第 2.3节）。

2.1 零知识证明

零知识证明使 证明者 能够向 验证者 证明者知晓某个计算陈述的正确性（例如，给定的布尔电路是可满足的），同时不会泄露有关其对该主张的证据的信息（例如，使该电路得到满足的见证）。存在用于证明知晓 NP 中所有问题的解的零知识协议 [26],；近年来，许多高效的协议已被开发并实现，用于一般陈述的零知识证明（例如， [8, 10, 12, 14–16、22、24,28,29,31,32,38,43,51]）。

其中一些研究专门针对有关程序正确执行的陈述，这些程序运行在包含随机存取存储器（RAM）的通用架构上，程序以低级机器码或高级语言表示 [9,10,12,14,16,17,22,27,29,31,32,38,51,52]。

我们的编译器结合了逐步 CPU 模拟与 TinyRAM [10–12]，所采用的类似方法：使用一种类 MIPS CPU，模拟可访问随机存取存储器的 C 程序及类似的低级语言程序。给定一个公开的 TinyRAM 程序、要模拟的执行步骤数上限，以及一个私有的程序输入，TinyRAM 编码器会生成一个 R1CS，输入的编码可使该 R1CS 得到满足。该约束系统由以下两部分组成：(1) 一组约束系统，用于验证每一步中完全基于寄存器的计算；(2) 一种新颖的内存检查子电路，它使用置换网络验证随机存取存储器操作的正确性。这种 CPU 展开技术非常适合支持数据依赖循环、控制流和自修改代码等语言特性。这种技术还可以自然地利用编译器前端和库等现有工具。

我们将 TinyRAM 式模拟与将程序块直接编译为电路门相结合 [17, 24, 52]（第 4节）。编译器的输出是一个电路，其可满足性等价于源程序中存在漏洞；其结构既不会泄露漏洞本身，也不会泄露触发漏洞的方式。

在我们的评估中，底层零知识协议是用于证明电路可满足性的 Mac'n'Cheese [9] 协议，由 Swanky [23] 库实现。这是一种交互式协议，证明者和验证者通过多轮通信来评估电路。最终，验证者会获知该电路接受了证明者提供的秘密见证（仅此而已）。

2.2 信息流

我们工作的一个核心贡献，是提出了一种实用方案，可以用零知识证明程序会泄漏数据。我们已将其用于证明，OpenSSL 的早期版本会在 Heartbleed 漏洞触发时泄漏私人数据（见第 5.2 节和第 6.3节）。该方案的设计要求对信息流进行形式化处理：具体而言，这种处理必须足够形式化，以便我们能够生成逻辑电路，且只有能证明存在泄漏的见证才能满足这些电路。为节省篇幅并保持清晰，我们将省略对完整编程语言中的信息流和泄漏的定义，但会充分详细地介绍我们的定义，以阐明其中的关键挑战和方法。

标记 L 是程序输入变量 I 的一个子集，这些变量被指定为私有输入；同时，L 也是输出变量 O 的一个子集，这些变量被称为公开输出。如果任意一对输入仅在私有输入上有所不同，而其在所有公开输出上的结果值均相同，则程序 P 满足关于 L 的非干扰；如果程序 P 不满足关于 L 的非干扰，则称其关于 L 存在泄漏。根据上述定义，可以通过一对执行来证明存在泄漏：这对执行仅在带有 L 标签的输入上有所不同，却产生了不同的带有 L 标签的输出。

非干扰有着精确而易于理解的形式化定义，能够描述某些关键软件的信息流要求 [20],，但它在实践中的局限性众所周知 [39, 40, 45]：实际程序的完整信息流规范往往并非非干扰属性，直观地说，这是因为处理敏感输入的程序通常确实需要泄露其中的一些信息；即便所需的信息流属性属于非干扰属性，要证明程序满足该属性，通常也需要仔细推理无界数据和控制。大量既有研究 [13, 18, 19, 25] 探讨了非干扰的推广形式，其中涉及可观察事件上的等价关系，以及丰富的编程语言和类型系统，并尝试证明这些推广形式成立。然而，非干扰属性仍然构成程序完整信息流要求的一部分，遗憾的是，这些要求既至关重要，又在实践中遭到违反（Heartbleed 就是一个典型例子）。这种情况印证了当前工作的主要关注点：证明非干扰属性遭到违反。

2.2.1 带标签的程序与执行

在最一般的形式下，信息流和泄漏都是针对执行对定义的。实用的程序监控器 [20, 49] 和类型系统 [39] 通过为程序的数据和控制结构标注元数据，并在执行过程中跟踪这些元数据，来证明关于所有执行对的性质。程序员或自动化分析都可以采用这些方法，直接为程序或执行添加注释。然而，正如下文所示，在我们的漏洞证明语境中，所需的保证与这些方法通常的应用场景有所不同。

从高层次来看，动态信息流监控器提供的保证如下。对一个程序执行的 n 个步骤进行标记，就是将每个程序变量在每个步骤 $0 \leq i < n$ 的值赋予一个敏感度标签。如果某种标记能够过度近似信息流，那么对于任何一对起始状态仅在高敏感度输入上有所不同的执行，程序产生的结果也只会在带时间戳的存储单元中具有高敏感度标签的单元上有所不同（静态分析和类型系统将这一性质扩展为：对于所有仅在敏感输入上有所不同的执行对，该性质都成立）。

这种过度近似在识别信息流时允许出现"假阳性"。例如，在只有输入 x 属于敏感信息、而返回值是公开信息的情况下，以下函数 always_true 不会泄露有关其输入 secret 的任何信息，因为对于每个输入值，它都返回 true：

```
布尔值 始终_为真 (布尔值秘密) {如果
( 秘密 ) 返回 秘密 ; 否则返回 ! 秘密 ; }
```

然而，许多自然的污点分析会将返回值标记为敏感，因为它是根据 secret 计算得出的。

对潜在泄漏进行过度近似，通常仍有助于程序员确保其程序不会泄漏：错误地判定安全程序可能发生泄漏或许会造成困扰，可能需要加以缓解以确保实用性，但在一定程度上可以接受。

然而，在我们证明零知识中的泄漏这一设定下，验证者仅了解到某个程序可能存在泄漏是不可接受的。我们的核心目标是证明它确实会泄漏（给定所谓的漏洞利用程序）。因此，我们将创建一种欠近似的标记，即当标签如此表明时，确实存在泄漏。接下来还需要通过实证表明，对于我们所关注的漏洞，这种标记确实能够检测出泄漏。

2.3 部分求值

在许多实际场景中，程序在编写和编译完成后，可能会在不同时间接收其输入的不同子集：例如，程序安装后，可能会加入一个配置文件；对于随后从网络接收的不同输入，该配置文件在所有执行过程中都保持不变。一个自然的目标是：给定一个程序以及其输入中可以固定的子集，生成一个专门化程序，以更高的性能处理剩余输入。更准确地说，对于输入变量为 X 和 Y 的程序 P(X,Y)，以及赋值 A：X $\rightarrow$ Words，将 P 在该赋值上的部分求值定义为程序 P_A，使得对于每个赋值 B：Y $\rightarrow$ Words，都有 P(A,B) 与 P_A(B) 相同。

在实用语言中对程序进行部分求值会带来若干复杂问题 [35]；其底层技术可概括为：**(1)** 在符号状态下执行程序，在该状态中，寄存器和内存地址可以映射到内存地址，也可以映射到由符号变量定义的项，而这些符号变量表示未知值；**(2)** 利用计算得到的符号状态（它描述每个控制点上的所有可能状态），简化各控制点处的控制结构。这种技术的不同变体可视为对常规优化编译器中实现的常量传播分析和常量折叠变换的激进扩展与推广[7]。

3 CHEESECLOTH 实现

CHEESECLOTH 可针对现实世界的漏洞生成零知识证明。它以公开的 LLVM 程序为输入（通常由 C、C++ 或 Rust 编译而来）；作为证明者运行时，还会输入一个能够触发程序中漏洞的秘密漏洞利用程序。CHEESECLOTH 输出一个零知识电路，用于验证程序的执行轨迹，并检查执行过程中是否发生了漏洞。该流程使证明者能够向验证者证明程序中存在漏洞，同时对漏洞及触发漏洞的漏洞利用程序保密。

CHEESECLOTH 以多种标准表示形式生成零知识电路，包括一阶约束系统（R1CS）[10] 和 SIEVE IR [6]。由于这些电路采用标准化格式进行序列化，CHEESECLOTH 不受所应用零知识协议的限制。作为证明者运行时，CHEESECLOTH 会输出该电路随附的见证。

CHEESECLOTH 可扩展为检查程序执行的不同属性。用户可以通过向编译流水线提供不同的输入标志，选择性地启用要运行的扩展。这些扩展实现了第5 节所述的内存和信息泄露漏洞检测检查。本节介绍 CHEESECLOTH 编译流水线的基线设计，其中包括：(1) MicroRAM 汇编语言，(2) MicroRAM 编译器，以及 (3) 见证检查器生成器。第4 节介绍了该设计的优化，使其能够扩展到真实世界的漏洞。

3.1 MicroRAM

MicroRAM 汇编语言对 CHEESECLOTH 至关重要。它是 CHEESECLOTH 操作所基于的核心中间表示语言，也是 MicroRAM 编译器将 LLVM 程序编译为的目标语言。见证检查器生成器会生成零知识电路，用于根据 MicroRAM 架构验证程序执行。

MicroRAM 深受 TinyRAM 的启发 [10, 11]，TinyRAM 是一种实用且高效的汇编语言，具有简单的转移函数，非常适合进行零知识执行验证。下面我们将介绍 MicroRAM 及其架构，并在第 3.1.1 节中准确描述其设计与 TinyRAM 的差异。

MicroRAM 是一种随机访问机，旨在高效检测程序执行中的漏洞。它是一种采用 Harvard 架构和按字节寻址的随机存取存储器的精简指令集计算机（RISC）。

MicroRAM 指令相对简单，包括 4 种布尔运算、8 种用于有符号和无符号整数的算术运算、2 种移位运算、5 种比较运算、2 种移动运算、3 条跳转指令、2 种用于读写内存的操作，以及 1 种返回结果并停止执行的应答操作。MicroRAM 机器不直接支持浮点运算和向量运算，必须通过软件实现。指令接收两个寄存器和一个操作数（寄存器或立即数）作为参数。例如，指令 `xor ri rj 255` 会将寄存器 `ri` 写为寄存器 `rj` 与立即数 `255` 进行异或运算的结果。CHEESECLOTH 等扩展可以像第 5 节所述，根据需要引入其他指令。

MicroRAM 机器的状态由程序计数器（pc）、k 个 64 位寄存器、由 2^{64} 64 位字组成的内存、一个用于指示截至目前的执行是否有效的标志（inv_flag），以及一个用于跟踪是否发生漏洞的标志（vuln_flag）构成。CHEESECLOTH 扩展也可以扩展 MicroRAM 机器的状态。

为了证明程序中存在漏洞，证明者必须提供一个秘密输入，使其产生一条有效的执行轨迹并触发漏洞。形式上，给定一个 MicroRAM 程序 P 和初始内存 m_0，如果在程序执行轨迹的最终 MicroRAM 状态中 inv_flag 为 false 且 vuln_flag 为 true，那么 P(m_0) 就证明了在 T 个步骤内存在漏洞。如果用于验证程序执行的任何检查失败，inv_flag 就会被设为 false。实现漏洞检测检查的扩展会在程序执行期间发现漏洞时，将 vuln_flag 设为 true。

3.1.1 超越 TinyRAM

如上所述，我们的 MicroRAM 机器以 TinyRAM 为灵感。这里我们介绍 MicroRAM 的设计与 TinyRAM 模型的不同之处。

• MicroRAM 的内存模型是按字节寻址的，而 TinyRAM 则是按字寻址的。按字节寻址的内存对于支持字符串操作和打包结构体等功能必不可少，而且无需添加子程序来访问完整字中的字节。

• TinyRAM 通过输入带接收输入。在 MicroRAM 中，输入直接传入内存，从而节省了 TinyRAM 将输入复制到内存所耗费的大量周期。不过，MicroRAM 程序可以通过多种方式请求非确定性提示，证明者无需预先将提示写入带并作出承诺，而是在收到请求时提供提示。这种方法更适合支持能够利用并行性或流式处理的后端，同时还能生成更小的电路。

• TinyRAM 使用 1 位条件标志进行分支，而 MicroRAM 不使用条件标志。这很有优势，因为 MicroRAM 面向多种后端，包括非布尔算术电路；在这些后端中，条件标志的开销高于普通寄存器[1]。此外，不使用条件标志的语义要简单得多，因此编译器、解释器和电路生成器也都更简单。我们发现，即使目标是布尔电路，使用条件标志带来的好处也抵不过它增加的复杂性。

• 我们尚未探索为 MicroRAM 使用冯·诺依曼架构 [12]，因为尽管这种架构具有渐近优势，但指令提取电路目前还不是零知识证明语句中的限制因素。

3.2 MicroRAM 编译器

MicroRAM 编译器实现为 LLVM 后端，以 LLVM 中间表示程序作为输入，并生成 MicroRAM 汇编作为输出。目前，我们通过使用 Clang 和 rustc 编译器前端将 C、C++ 和 Rust 程序编译为 LLVM 中间表示，从而支持这些程序。未来，通过连接相应的 LLVM 编译器前端并编写相应的标准库，还可以添加对 C#、Haskell 或 Scala 等其他语言的支持。

我们的编译器后端支持 LLVM 中间表示语言的很大一部分。编译器支持不同位宽整数的所有布尔运算和算术运算、位运算、所有非并发内存操作（包括使用 getelementptr 进行的指针运算）、转换操作、函数调用、可变参数、比较操作和 phi 节点。浮点运算等复杂操作通过 LLVM 编译器遍以软件方式实现。

不支持异常及所有异常处理指令；但只要证明者披露感兴趣的执行（该执行会触发漏洞）不会抛出任何异常，我们仍然可以容忍包含异常的程序。这是因为 MicroRAM 编译器会将所有异常处理指令转换为陷阱，通过设置`inv_`标志使执行轨迹无效。通过插入陷阱，MicroRAM 编译器可以处理包含任意数量不受支持特性的程序，只要证明者愿意揭示这些特性未涉及易受攻击的执行。借助这一简单技巧，用户无需手动移除不受支持的特性，即可编译现实世界中的程序。启用陷阱后，证明者必须注意不要泄露过多有关底层漏洞的信息。第 3.2.3 节将更详细地讨论证明陈述如何泄露有关其见证的信息所带来的安全性影响。

3.2.1 标准库

MicroRAM 支持 C 标准库和 POSIX 系统调用中的大部分功能，并使用 Picolibc [4]：这是一个提供 C 标准库 API 的库，最初专为 RAM 容量有限的小型嵌入式系统设计。Picolibc 支持多种广泛部署的目标架构，包括 ARM、RISC-V 和 x86-64。

我们将 MicroRAM 实现为 Picolibc 的一种目标架构。这使 MicroRAM 编译器能够支持大部分 C 标准库和 POSIX 系统调用。此外，这样做也很方便，因为它允许证明者公开定制系统调用的行为。例如，在我们对 OpenSSL 的案例研究中，受害者服务器通过网络接收攻击者发送的恶意请求。我们定制了read 函数在本机编译时的行为，以拦截并记录通过网络接收的所有数据。编译为 MicroRAM 时，read 函数会返回之前记录的漏洞利用请求，该请求从秘密内存中加载。我们还定制了`malloc`和`free`的实现，以便高效检测内存漏洞（第5.1.1节）。

3.2.2 生成辅助信息

正如我们将在后续章节中看到的，CHEESECLOTH 需要非确定性建议，以便高效生成零知识电路，验证一次执行中的内存一致性（第 3.3 节）以及漏洞是否存在（第 5.1 节）。为帮助证明者生成这些建议，MicroRAM 编译器会运行两遍解释器。第一遍在没有任何建议的情况下执行程序，并记录所需的建议。第二次执行采用非确定性语义并记录执行轨迹，随后将其传递给见证检查器生成器，为证明者生成见证。

[1]如果完整的字能够容纳在一个域元素中，那么该标志的大小就与寄存器相同，但需要特殊电路，并且限制更多。

3.2.3 安全性

MicroRAM 生成零知识证明，确保不会泄露有关见证的额外信息。然而，证明陈述本身可能会泄露有关秘密输入的信息。例如，在 MicroRAM 和 TinyRAM 中，电路会揭示执行长度的时间上限T。在 Pantry/Buffet 中，电路会披露执行中每个循环（以及递归函数）的上限T_i。在 vRAM 中，执行期间运行的每条指令都会 [53], 向验证者公开。我们认为，有必要对这种信息泄露进行形式化。未来一项有趣且重要的工作是定义一个形式化框架，用于分析这些编码的安全性。

3.2.4 公共输入预处理

激进优化的一个机会，是对由程序的公共输入决定的逻辑进行公开求值。许多实际程序会从多个来源收集输入，其中一些并非秘密（即与漏洞无关）。如果证明者和验证者在定义证明陈述时达成一致，认定只有部分输入属于敏感秘密（例如，从网络连接接收的数据包），而其他输入并非如此（例如，简单的配置选项），那么就可以立即优化最终的证明陈述：生成证明陈述后，对所得电路中对应于非敏感输入的输入线进行部分求值。

CHEESECLOTH 通过一个编译器遍来支持此类情况，该遍会确定程序中不依赖秘密输入的最长前缀。随后，MicroRAM 编译器将公共前缀与程序的其余后缀分离，并分别编译。当证明者和验证者运行解释器时，解释器会执行前缀，并将执行后得到的状态定义为公共快照，其中包括寄存器和内存。证明者运行解释器时，接着会使用该快照和其私有输入执行剩余的后缀，以生成证明陈述的见证。在实践中，这项简单的优化效果显著，将 OpenSSL 零知识证明陈述中的执行步骤数从 25M 降至 1.3M （第 6.3 节）。

该编译器优化实现了相对受限形式的部分求值和常量折叠（第2.3节）。我们的初步经验表明，进一步扩展可大幅提升 CHEESECLOTH的性能：一个关键的技术挑战在于，程序在整个执行过程中可能会对公共数据进行大量处理，但这些处理往往与针对敏感输入的计算交错进行。原则上，对公共计算中每个交错阶段进行求值是可靠的，但要实现自动化，只能确保公共阶段和秘密阶段使用的存储区域互不重叠。此类自动化有望通过应用指向分析和形状分析来实现，[46–48], 包括分离逻辑 [44]。

3.3 WitnessChecker 生成器

见证检查器生成器以 MicroRAM 程序为输入，并生成零知识电路，再以 R1CS 和 SIEVE IR 等标准化格式进行序列化。它还接受非确定性辅助信息作为输入，并在作为证明者运行时输出该电路的秘密见证。

WitnessChecker 生成器为素数域 $2^{128} - 159$ 构建算术电路。作为一项优化，它会自动对与秘密输入无关的门进行常量折叠。为支持大型电路并避免内存耗尽，它会将电路序列化数据流式写入文件。该流式处理与秘密见证无关，因此为证明者和验证者生成的电路相同。

WitnessChecker 生成器接受的非确定性辅助信息，提供程序执行情况的描述，以及运行程序所需的辅助信息。具体而言，对于包含 T 个步骤的执行，辅助信息包括初始程序内存、构成执行轨迹的 T 个 MicroRAM 状态，以及从步数到各步骤所提供的额外辅助信息的映射。这些额外辅助信息包括用于指明内存读取或写入内容的内存端口，以及指示执行应在当前步骤暂停的停顿标记。

见证检查器生成器生成一个零知识电路，用于验证见证是否描述了该程序的有效执行轨迹，以及是否发生了漏洞。该电路分为四个关键部分：(1) 转移函数电路，(2) 内存一致性电路，(3) 状态转移网络，以及 (4) 公共程序计数器片段。我们在此介绍前两个部分，它们的结构与 TinyRAM [10]的电路构造类似。后两个部分将在第 4.1节介绍。

转移函数电路。转移函数电路会检查单步执行。这些检查串联起来，以验证整个执行轨迹。图 1 展示了转移函数电路的伪代码。它以当前 Micro-RAM 状态、下一状态的电路线表示，以及当前步骤所需的任何额外辅助信息作为输入。随后，电路根据程序计数器获取要执行的指令，并通过索引访问机器寄存器，提取该指令的参数值。它通过对指令进行多路复用，计算该步骤的预期结果。最后，电路确保计算出的预期状态与作为辅助信息提供的下一状态相匹配。

内存一致性电路。内存一致性电路与 TinyRAM 的电路相似，但地址采用按字节寻址，而非按字寻址。每一步都可能有相应的内存端口辅助信息，说明地址以及从内存中读取或写入的内容。转移函数电路会验证执行轨迹是否与内存端口辅助信息相符。所有内存端口都按地址和步数排序。内存一致性电

```
函数转移_函数（电路，当前_st，下一个_状态）{
  令 预期_状态 = 当前_状态.克隆();
  令 指令 = 获取_指令（电路, 当前_状态.程序计数器);
  令 参数1 = 索引(circuit, current_st.regs, instr.op1);
  令 参数2 = 索引(circuit, current_st.regs, instr.op2);

  令 结果 = circuit.mux(instr.opcode == 异或,
                异或(电路, arg1, arg2), ...);
  预期_st.pc = circuit.mux(
                是_jump(circuit, instr.opcode),
                结果，
                电路.add(current_st.程序计数器, 1));
  write_索引(电路, expected_st.寄存器, instr.dest,
             结果);

  电路.assert(expected_st == next_st); }
```

图 1：用于验证单个 MicroRAM 步骤的转移函数电路伪代码。

路会线性扫描内存端口，以确保对给定地址的所有读写操作都与之前的内存操作一致。例如，read 函数应返回先前写入该地址的相同值。最后，内存一致性电路会检查，排序后的内存端口是否是转移函数电路所用内存端口的一个排列。第 5.1 节将介绍如何改进这些检查，以便高效检测内存漏洞。

4 优化

本节介绍了 CHEESECLOTH的两项关键优化：构造程序计数器公开的执行轨迹（第 4.1 节），以及根据指令稀疏性调整步骤（第 4.2 节）。第 6.4 节对这些优化的效果进行了实证评估。

4.1 公共程序计数器片段

MicroRAM 机器旨在尽可能缩小用于检查转移函数电路的电路规模。然而，即使 MicroRAM 的指令集很小，转移函数电路仍然很大。这是因为每个转移函数都必须支持每一步执行中的所有操作。如果我们能够移除所有未使用的功能呢？这正是 vRAM [53],所采用的方法：调整电路，使其检查每一步执行的指令。由此生成的电路要小得多，但遗憾的是，已执行指令的执行轨迹也会被公开。内存和寄存器中的值仍会保密，不过验证者很容易就能发现程序中的脆弱代码位于何处。本节将介绍公共程序计数器片段，它们能生成小得多的电路，同时不会泄露执行轨迹。

公共程序计数器片段是由一系列转移电路组成的序列，其中程序计数器是硬编码且公开的。利用常量折叠，公共程序计数器片段中的所有指令取值都已知，因此可以移除每一步中未使用的功能。例如，程序计数器公开后，图1_中的指令取值以及随后针对该指令的多路复用器操作都可以通过常量折叠消除。我们会为程序中的所有直线代码片段生成公共程序计数器片段，并实现了一个编译器遍，利用朴素的控制流分析来估算每个片段会被使用多少次。该分析会接收一个全局界限，用以指定循环展开次数；同时，它通过统计程序中某个函数的调用点数量，来估算该函数会被调用多少次。

为确保执行轨迹的安全性，片段的周期计数器保持私密。此外，我们引入状态路由网络，使一个片段的结束状态可以作为电路中任意其他片段的初始状态。与内存路由网络一样，状态路由网络的路由信息由证明者提供并保密。

作为进一步的优化，我们会尽可能避免使用状态路由网络。例如，当公共程序计数器片段分支到两个静态已知的位置时，我们会将该片段的结束状态直接连接到代表这两个位置的片段。

循环展开的界限可能不足以支持某些执行，因此，对于某段代码，流水线生成的公共程序计数器片段可能不够。作为备用方案，流水线还会生成私有程序计数器片段，它们与公共程序计数器片段类似，只是不公开程序计数器。私有程序计数器片段看起来类似于规模小得多的 TinyRAM 电路，不同之处在于其起始和结束状态来自状态路由网络。这些片段的电路规模显著更大，但可以在执行过程中的任意时刻执行程序的任意部分。

4.2 稀疏性

采用第 3.3 节所述的朴素中央处理器展开方式时，每个转移函数都必须包含一个内存端口，这会导致内存一致性网络以 O(T log T) 的速度增长，其中 T 是已执行的步骤数。不幸的是，其中大多数门都被不访问内存的执行步骤浪费了。CHEESECLOTH 通过移除部分未使用的内存端口来缓解这一问题，从而缩小内存一致性电路的规模。

这项优化的关键观察是，内存操作很少连续发生。即使程序执行内存密集型操作，内存指令之间通常也会穿插其他指令。例如，对缓冲区中的值求和时，在两次内存读取之间，需要执行若干步骤来递增指针并相加这些值。这使我们能够让 s 个连续步骤共享一个内存端口，从而将内存一致性网络缩小

s 倍。

我们将内存稀疏度 s 定义为共享单个内存端口的步骤数。CHEESECLOTH 根据对代码进行的静态分析来选择 s。该分析会确定任何可能的执行中，两次内存操作之间的最小距离。对于静态上未知的跳转（例如，通过指针解引用调用函数），该分析会直接将控制流可能跳转到的所有指令都纳入考虑。随后，MicroRAM 编译器和见证检查器生成器会使用这个内存稀疏度数值来生成优化后的电路。

给定内存稀疏度 s，见证检查器生成器会将连续的 s 个步骤分为一组，并为这组步骤创建一个内存端口。多路复用器会将这个内存端口连接到整组步骤，并利用非确定性辅助信息将结果发送到正确的步骤（如果有）。

如果 s 大于执行轨迹所体现的实际稀疏度，那么（运气不佳时）多个内存访问可能落入同一组步骤，而该组只有一个内存端口。CHEESECLOTH 通过插入停顿指令来处理这种情况：这些指令会延迟内存操作，直到它们被推迟到下一组拥有独立内存端口的步骤中。插入停顿指令可能代价高昂，但缩小内存一致性电路的规模带来的收益更大（第 6.4 节）。未来的工作中，我们将探索交换程序指令，以减少停顿指令，并确定对大多数程序而言最优的 s 参数。

```
void* malloc(size_t size) {
  // 从建议值中获取指针 char* addr = __cc_
  malloc(size); /* 计算并验证证明者提供的 * 分配大
  小。 */ size_t region_size =1ull << ((addr >> 58)
  & 63); /* 分配的区域必须有足够空间容纳 `size` 字
  节，此外还要为元数据留出 * 一个额外的字。 */__
  cc_valid_if( region_size >= size +
  sizeof(uintptr_t), "分配的区域大小过小"); /* `
  region_size` 始终是 2 的幂，且至少与 * 字大小相
  同，因此地址必须是字大小的 * 倍数。 */__cc_
  valid_if(addr% region_size == 0, "分配的地址未
  按其区域大小" "对齐"); /* 将 1（已分配）写入元数
  据 * 字段，并将其设为 poison 以防篡改； * 如果元
  数据字已经被设为 poison，则使执行轨迹 * 无效
  （证明者试图为两次独立的 * 分配返回同一区域时就
  会发生这种情况）。 */ uintptr_t* 元数据 =
  (uintptr_t*) (addr + region_size - sizeof(
  uintptr_t));__cc_write_and_poison(metadata,
  1); // 后续计算..... return (void*)addr; }
```

图 2：非确定性 malloc 的实现。

5 漏洞编码

本节介绍 CHEESECLOTH 如何编码两类普遍且关键的软件漏洞：内存不安全（第 5.1节）和数据泄漏（第 5.2节）。

5.1 内存不安全

现在，我们介绍 CHEESECLOTH 如何高效地对内存建模并表示内存漏洞。在 CHEESE-CLOTH 中，内存是一个由 2^{64} 字节组成的数组，其中一半为堆保留，剩余部分用于全局变量和栈。我们的方法是跟踪有效内存（例如，已分配的数组），并在程序访问无效内存时报告漏洞（即设置 bug_标志）。在执行开始时，唯一有效的内存是存储全局变量的区域；执行期间，malloc 使已分配的区域变为有效，而 free 再将其变为无效。通过这种技术，我们可以捕获以下内存错误：

• 未初始化访问。所有未初始化内存均无效，因此任何使用都会触发错误。

• 释放后使用。当一个区域被 free 后，它就会失效，因此任何对它的使用都会触发错误。

• 重复释放。free 的实现首先从待释放区域中 read 函数读取一个字，前提是该区域尚未有效，就会触发错误。

• 越界访问。如果程序访问了一个越界地址，那么这个新位置可能（见下文）无效，从而触发错误。

显然，进行此类边界检查的常规执行可能会漏掉越界访问错误：访问恰好落在另一个有效区域时，就会发生这种情况；此外，如果中间的 malloc 在错误触发前使该区域变为有效，也可能漏掉 free 后再次 free 或释放后使用错误。然而，我们只需证明某次执行中存在该错误，因此我们实现了一个由非确定性辅助信息引导的 malloc；这使证明者能够选择内存分配布局，以确保触发该错误。

虽然这里介绍的技术专门针对堆内存错误，但同样的思路也可应用于栈。

5.1.1 对动态内存分配进行编码

具有非确定性的 malloc 实现本身就带来了挑战。如果不加以约束，证明者可能伪造一次会触发虚假错误的执行。例如，证明者可能分配相互重叠的区域，使得其中一个

被释放而另一个被访问时触发虚假错误。因此，我们对 malloc 和 free 的实现（图 2 ）着重验证这些非确定性选择是否合法。如果检测到违规行为，该执行将被标记为无效，并带有`inv_flag`，不会被验证者接受。

为确保 malloc 永远不会返回重叠区域，我们预先确定了不同大小的、经过对齐且互不重叠的区域，供 malloc 选择。具体来说，我们将内存划分为 2^6 个大小为 2^{58}，的池，然后将第 i 个池进一步细分为大小为 2^i 的区域。malloc 会将请求的大小向上取整到下一个二的幂，然后返回该大小的一个未分配区域的起始位置。例如，malloc(15) 必须返回第 4 个池中的一个区域，并按 16 字节对齐。事实上，只需查看返回的指针，我们就能轻松验证 malloc 是否分配了正确的区域：前 6 位确定池，剩余部分确定对齐方式。

最后，malloc 不能在同一个指针尚未被释放的情况下再次返回它。为此，我们在每个区域中增加一个字，用于存放元数据；当该区域被分配时，这些元数据会被标记并置为无效。如果该区域再次被分配，无效的元数据会再次被置为无效，从而通过设置 inv_flag 使执行轨迹无效。

此外，能够跟踪所有内存位置有效性的 malloc/free 实现会非常低效。幸运的是，证明者确切知道错误会在哪里发生，因此 malloc/free 实现只需跟踪该位置的状态。在执行开始时，证明者会对存储在全局变量 __cc_memory_error_address 中的一个秘密位置作出承诺，然后 malloc/free 只跟踪该位置的有效性。具体而言，如果分配/释放的区域不包含 __cc_memory_error_address，那么 malloc/free 就不会检查错误，并且会以常数时间运行。

5.2 数据泄漏

证明泄漏的一种直接方法，是在零知识电路中直接编码非干扰的定义。具体而言，可以验证两次程序执行：两次执行中只有敏感输入不同，但公开输出也不同。然而，这种方法所需陈述的规模将是验证单次执行所需规模的两倍。因此，我们或许可以尝试通过单次执行来证明泄漏，并为存储标注标签（第 2.2 节）。然而，这类系统传统上只能用于证明程序 可能会 泄漏信息；若不直接提供一次违规执行，这就无法令人接受地确切证明存在泄漏（第 2.2.1 节）。

指定泄漏 为了识别敏感源和汇，我们将 source 和 sink 指令加入 MicroRAM 指令集，并分别由用户级函数 taintSource 和 taintSink 直接封装。source 用于标注给定字节的数据包含敏感数据；sink 用于标注给定字节被输出到通道。

针对这个扩展后的指令集架构，将信息流和泄漏的一般定义（第 2.2 节）实例化后，如果某个 MicroRAM 程序存在两次执行，其输入仅在传递给 source 的地址处不同，却导致传递给 sink 调用的某个地址处的值不同，那么该程序就会发生泄漏。证明者与验证者协作扩展主体程序，使其调用 taintSource 和 taintSink，以标注敏感源和汇，从而证明泄漏。

证明泄漏 为了可靠且精确地证明泄漏，我们提出了一种新颖的标记系统，用于跟踪程序存储可能或必然持有的秘密。共有四种标签，记作以下形式，并构成偏序：

$$\bot \sqsubseteq \ell_0, \ell_1 \sqsubseteq \top$$

其具有一个最小上界（即并）运算，记作 $\sqcup$。标签 ℓ_0 和 ℓ_1 表示数据必定属于两个主体之一； $\top$ 表示数据的敏感性未知； $\bot$ 表示数据不得受某个主体影响。采用此标记方案时，当 $\ell \neq \top \wedge \ell \not\sqsubseteq \ell_c$，写入带有 ℓ_c标签的接收端的 ℓ标签数据必定会发生泄漏。

MicroRAM 状态经过扩展，使每个寄存器和每个内存字节都关联一个标签，这与以往的泄漏监控器类似 [20, 42, 49, 50]。另外两种标签用于模拟寄存器算术以外的指令所产生的影响。控制上下文标签 γ 始终保持为 $\bot$，表示程序执行尚未因秘密数据而发生分支；否则为 $\top$。同样，存储上下文标签 σ 始终保持为 $\bot$，表示程序尚未向秘密地址写入数据；否则为 $\top$。

每次赋值 x:=e 都会将 x 的标签设为 L(e) $\sqcup\gamma\sqcup\sigma$（其中 L(e) 是 e 的标签，定义如下）；因此，如果程序曾根据秘密数据进行分支，或写入秘密地址，x 的标签就会设为 $\top$。如果 e 是算术/逻辑运算 f (y)，那么当 L(y) 为 $\bot$ 时，L(e) 为 $\bot$，否则为 $\top$；若 f 是双射，则 L(e) =L(y)：我们当前的实现采取保守策略，仅将单寄存器表达式（即复制源）标记为 L(y)。如果 e 是加载 *p，那么 L(e) 为 L(*p)。条件分支会根据标签的说明更新 γ，内存存储则会更新 σ；由于篇幅限制，这里省略形式化说明。

许多自然程序会发生泄漏，但该标记系统无法证明这些程序确实如此，这可能是因为泄漏发生在分支操作之后，或发生在向某个带有 $\top$ 标签的值存储数据之后；也可能是因为某个秘密值通过一个未被识别为双射的操作进行了传播。这类情况限制了标记方案可用于证明泄漏的场景，但当该方案声称某个给定程序会发生泄漏时，并不会威胁其有效性。未来的工作可以通过使用有效的逻辑公理来细化指令解释，从而处理这些情况（例如，对于每个值 x，都有 $x + 0 = x$ 这一事实）。这类威胁及其缓解措施，与对精确性的威胁以及在证明程序执行可能发生泄漏时进行的潜在细化，是相互对应的。

6 评估

我们通过三个案例研究来评估 CHEESECLOTH ，这些案例研究展示了现实世界软件漏洞的零知识证明。漏洞的规模会随代码大小和执行轨迹长度而变化，以展示 CHEESECLOTH 的能力。我们还对优化进行了基准测试（第 4 节），以评估其有效性。

表。1 展示了使用 CHEESECLOTH 为我们的案例研究生成零知识证明的结果，这些案例研究包括 GRIT、FFMPEG 和 OpenSSL。对于每个案例研究，我们报告程序的大小（以 Micro-RAM 指令数量衡量）、证明漏洞所需的执行步骤数，以及最终零知识电路中的乘法门数量。我们使用 Mac'n'Cheese [9] 交互式零知识协议（由 Swanky [23] 库实现）证明零知识电路的可满足性。我们记录协议运行时间，以及证明者与验证者之间的通信成本。所有测量均在一台运行 Debian 11 的 128 核 Intel Xeon E7-8867 中央处理器（配备 2 TB 随机存取存储器）上进行，不过我们的实现通常使用的内存要少得多（表 1）。

6.1 GRIT 中的内存不安全

GBA 光栅图像转换器（GRIT） [34] 会将位图图像文件转换为 Game Boy Advance 可读取的图形格式。位图图像包含标头、用于指示图像中颜色的调色板数组，以及图像的像素。对于 24 位色图像，GRIT 解析器假定调色板大小为零，并分配不包含调色板空间的缓冲区。在填充缓冲区时，它会检查图像标头中的调色板条目数量，却没有检查该数量是否与分配时采用的假定调色板大小一致。因此，格式错误的 24 位色图像可以在已分配的缓冲区之后写入任意量的数据（最多可达文件长度）。

为演示这一内存错误，我们构造了一个包含 0x3000 字节像素数据和 12 字节调色板数据的 24 位色漏洞利用图像。在 Linux 上，这 12 字节的溢出会覆盖堆元数据，并在内存分配器中触发断言失败。通过 CHEESECLOTH 运行时，我们生成一个零知识证明，证明在 GRIT 执行的六千个步骤内触发了内存错误，同时不会泄露触发该错误的图像或错误在代码中的发生位置。

6.2 FFmpeg 中的内存不安全

FFmpeg 是一款用于录制、转换和流式处理音频与视频的工具[2],，广泛应用于 Chrome、Firefox、iTunes、VLC 和 YouTube 等热门软件项目。FFmpeg 以 C 语言编写，长期受到漏洞困扰，这些漏洞会危及内存安全，使攻击者能够执行代码并通过网络共享本地文件。在 v1.1.2 之前的 FFmpeg 版本存在一个漏洞 [1]，该漏洞由函数 gif_copy_img_rect 中的内存错误引起。此函数用于在缓冲区之间复制 GIF 文件的帧。早期版本的 gif_copy_img_rect 会直接使用输入图像的高度，以不安全的方式计算内存缓冲区末尾的指针。攻击者可以精心构造 GIF 文件，利用这一计算方式诱使 FFmpeg 向数组边界之外的内存写入数据。

为了在零知识证明中证明 FFmpeg 存在内存不安全问题，我们手动制作了一张 GIF 图像，用于利用所述内存漏洞。我们将该图像和一个调用 FFmpeg 视频解码器的程序模块传递给 CHEESECLOTH，后者生成了一份越界访问证明。关于这个用于漏洞利用的 GIF，唯一被披露的事实，是它在执行的 7.6万步以内触发了越界访问这一事实所隐含的信息。

在公共输入上对 FFmpeg 进行预处理 由于可以对 FFmpeg 的证明语句进行激进优化，因此最终通过手动程序划分后应用 CHEESECLOTH 的常量折叠转换遍，实现了这一目标。之所以需要进行划分，是因为 GIF 模块中公共计算与秘密计算交错执行，其执行过程为：(1) 将给定的秘密 GIF 文件解复用为一系列数据包；(2) 使用公共配置设置初始化解码器状态；(3) 执行包含漏洞的编解码器。

尽管阶段（2）完全基于公共数据进行计算，但 CHEESECLOTH的常量折叠过程不会对其进行优化，因为该过程在检测到使用秘密数据的计算时便会停止，因此不会优化阶段（1）之后的任何程序片段。为解决这一问题，我们手动按阶段划分程序，分别对每个阶段应用 CHEESE-CLOTH的常量折叠过程，并将生成的优化后 MicroRAM 代码链接起来。总的来说，我们对 FFmpeg 的案例研究表明，有必要进一步研究和设计更激进的常量折叠过程，这类过程或许可以应用更复杂的静态程序分析（第 2.3 和 3.2.4节）。

6.3 OpenSSL 中的信息泄漏

OpenSSL [3] 是一种广泛部署的开源密码学库，其中实现了 SSL 和 TLS 协议。OpenSSL 1.0.1 至 1.0.1f 版本存在一个被称为 *Heartbleed*的严重漏洞，该漏洞于 2014 [5],被发现，远程攻击者可利用它，彻底泄漏该协议执行期间存储的信息，包括其他客户端的敏感信息和私钥。

本文不在此详述 SSL 和 OpenSSL；就我们的研究而言，只需指出，SSL 通信方支持多种请求，包括向另一方请求存储数据，以及请求对心跳信号作出回复：该信号仅由.....发送

程序	代码大小（千条指令）	执行步骤（千步）	乘法门（百万个）	协议时间	协议内存
GRIT	3	5	26.7	3分钟40秒	845 MB
FFmpeg	24	79	672.7	1小时22分钟	19 GB
OpenSSL	340	1,300	17,049.5	36小时45分钟	460 GB

表 1：针对每个案例研究生成并运行软件漏洞零知识证明的结果。

```
void 处理_心跳(SSLRequest*req) {无符号整数 长度 =
解析_心跳_长度(req);无符号字符 *心跳 = 获取_心跳
(req);无符号字符 *响应 =
malloc(len);memcpy(response,heartbeat,
len);write(response,len); }
```

图 3：展示 Heartbleed 漏洞的伪代码。

以获得响应，从而确保对方仍在响应。

心跳请求和响应对于 Heartbleed 漏洞的运行至关重要。格式正确的请求由数据缓冲区 d 和长度字段 n < |d|组成。对此类请求的正确响应应返回 d 中包含的前 n 个字节。然而，一方可能会发送格式错误的请求，其中 n > |d|。对此类格式错误请求的正确响应是拒绝该请求。

OpenSSL 的实现（如图 3 中的伪代码函数 process_heartbeat 所示3）未能正确实现协议的这一方面，而是返回了与输入缓冲区相邻的内存中的 n 个字节。 process_heartbeat 从客户端接收心跳请求，并将请求中提供的心跳字符串回显。它首先从客户端请求中解析出心跳字符串的长度，然后获取请求中指向该字符串的指针。接着，它分配一个响应缓冲区，并将心跳字符串中的 len 个字节复制到响应缓冲区，随后将其发送回客户端。由于 process_heartbeat 不会将提供的心跳长度与心跳字符串的实际长度进行核对，因此，如果声称的长度大于实际提供的心跳字符串长度，就会将客户端请求之外的内存内容发送回客户端。这种漏洞在实践中可以被利用，而且已有演示表明，它能够泄露内存中的敏感数据，例如加密密钥和密码。

我们使用 CHEESECLOTH 以零知识证明了，OpenSSL 1.0.1f 版本在执行 1.3M 个步骤后会泄漏任意用户信息，且数据仅通过寄存器复制、加载操作和存储操作进行传播；虽然该陈述揭示了执行泄漏所需计算量的上限，以及用于实施泄漏的指令类型信息（如下所述），但它并未直接指出处理心跳请求的函数缺少何种验证，也未说明心跳请求是否与泄漏有关。下文将更详细地介绍所证明的陈述，以及其中的技术挑战和解决方案。

```
int 登录_处理程序(
  SSL连接*c, char *密码,int len) {
  ...
  标签 l =getLabel(c);
  对于 (大小_t i = 0; i < len; i++)
    taintSource(密码 + i, l);
  ... }
int ssl3_写入(
  SSL连接*c, 字符 *buf, 整数 len) {
  ...
  标签 l =getLabel(c);
  对于 (大小_t i = 0; i < 长度; i++)
    污点汇(buf + i, l);
  ... }
```

图4：OpenSSL 函数 login_handler 和 ssl3_write 的版本，其中我们增加了用于指定信息源和汇的操作。密码被赋予当前连接的标签；如果写入网络的数据带有来自不同连接的标签，则会检测到泄漏。

指定 OpenSSL 的泄漏。我们工作的一个主要挑战是提供一种方案，用于识别敏感源和汇，使其满足以下条件：

1. 仅了解主体程序所处理数据的验证者，应能够检查修改后的程序，并明确得出结论：该程序正确地定义了信息源和汇。2. 为了定义信息泄漏源与汇而对程序所做的任何修改，都不得泄露有关触发该泄漏的输入的额外信息。

我们通过指定函数 taintSource 和 taintSink（第 5.2 节）来定义源和汇的敏感性。我们发现，这样的库很适合用于指定 OpenSSL 中的信息流；图 4 给出了对 OpenSSL 代码库中用于标记源和汇的 C 函数所做修改的伪代码。函数 login_handler 接收一个 SSL 连接 c 和一个缓冲区 password，假定其中包含 len 个将通过 c 传输的敏感信息字节，并将从 password 开始的 len 个地址标记为 c 的标签。函数 ssl3_write 接收一个 SSL 连接 c 和缓冲区 buf，假定其输出 len 个字节，并将输出通道上从 buf 开始的 len 个地址标记为 c 的标签，作为汇。

Th对 login_handler 和 ssl3 的修改 write_

程序	不含 公共程序计数器片段的乘法门（M）	不含 稀疏性的乘法门（M）
GRIT	42.3（37%）	27.9 (4%)
FFmpeg	716.9 (6%)	709.9 (5%)

表 2：禁用不同优化时电路中的乘法门数量，以及与表 1. 中基线数值相比的规模增长百分比。

这说明了一阶标签的实用性：它们可以通过操作收集和设置，而不是像某些操作那样，只将地址设为高源或低汇，即便是在只关注属于单个主体的信息的情况下也是如此。通过使用一阶标签，我们能够编写小型规范函数，仅统一网络连接与给定缓冲区之间的标签，然后在能够便捷获取连接及相关缓冲区的上下文中，简洁地修改原有程序逻辑。

证明 OpenSSL 的泄漏 适当修改 OpenSSL，使其调用 taintSource 和 taintSink 函数后，就可以通过生成一个陈述来证明其泄漏；该陈述的解对应于一次服务器执行：服务器运行 OpenSSL，并将敏感数据从一个连接泄漏到另一个连接。我们生成了这样一个陈述：服务器首先响应一个公开登录请求，其中密码被标记为敏感数据。随后，服务器处理一个秘密的恶意心跳请求，该请求返回了前一个请求所属连接中的密码。

我们使用 CHEESECLOTH，通过验证前文所述的执行轨迹来证明 OpenSSL 的泄漏；该执行轨迹源自其最初披露的某个漏洞利用程序。根据我们提出的必须泄漏标记方案（第 5.2 节），通过源和汇注释检测泄漏，而验证者只能获知恶意请求长度的上界。我们发现，该标记方案能够更加高效地证明泄漏，使总体电路规模相比双执行轨迹方法减少了 30.6%。CHEESECLOTH 用时约 37 小时证明了 OpenSSL 的漏洞，协议通信量为 460 GB。

6.4 优化

表 2 展示了我们的关键优化所带来的改进（第 4 节）。我们分别禁用了各项优化，运行了 GRIT 和 FFmpeg 案例研究，并报告了生成的零知识电路中的乘法门数量。此外，我们还提供了相较于表 1 中基线数据的改进百分比。public-pc 优化使较短的 GRIT 执行的门数量减少了 37%，使 FFmpeg 的门数量减少了 6%。尽管有所改进，这些结果表明，我们用于确定较长执行所需生成的公开片段数量的分析仍有改进空间。

稀疏性优化使 $= 2$ 的门数量 modest地减少了 4%–5%。

7 相关工作

近期工作首次迈出了在零知识中进行漏洞证明的令人振奋的步伐。 BubbleRAM [29] 是一个高效的漏洞证明框架，利用新颖的协议在算术域和布尔域中的计算之间进行转换，高效处理只读和读写存储器，并使用 Stack 协议 [30] 证明带有显式析取的电路满足性。尽管我们当前的陈述编译器与 BubbleRAM 存在部分重叠，因为它实现了用于建模 RAM 计算的较早方案[10], ，但本文的大多数关键贡献——即使用部分求值简化展开的计算，以及用于生成应用程序泄漏陈述的新颖方案——基本上独立于 [29, 30], 的贡献；我们认为这些方法可以组合使用。具体而言，Stack 在代表实际通用漏洞披露的代码片段上进行了评估，规模最高达 50 代码行数；由于其对析取的高效支持，它可以扩展到证明更多此类代码片段中的某一个存在漏洞，但如果某个特定代码段的规模增大，它很可能会从 CHEESECLOTH 的程序优化中获得显著收益。

Reverie [27] 是一个用于证明微处理器代码中漏洞利用程序的框架，由一个将给定程序编译为算术电路的电路生成器，以及对“脑内 MPC”协议 [33] 的一种实例化 [36] 组成。编译器会根据漏洞利用程序生成语句，这些漏洞利用程序被形式化为执行一条指定的指令，该指令会发出错误条件信号（即直接在程序控制流中形式化的可达性属性违规）；对 Reverie 的评估表明，它可用于证明在 MSP430 微处理器上需要最多 51K 个周期的夺旗赛（CTF）漏洞利用程序。Reverie 的核心贡献与 CHEESECLOTH 的贡献大体上互为补充；后者有望经过调整，高效地将有关中间语言程序的漏洞语句编译为控制可达性属性。

近期关于零知识静态程序分析的研究 [21] 提出了相关技术，可在不泄露程序更多细节的情况下，证明对所有程序执行的过近似，并基于一个用于污点跟踪信息流分析的抽象域实例化了该框架。静态分析本身旨在证明程序可能泄露信息：因此，它无法得出能直接表明程序必定泄露信息的结果，尽管在许多情况下，它可以提供证据，强有力地影响分析人员对程序实际上可能泄露信息的判断。

我们的 MicroRAM 机器受到 TinyRAM [10] 的启发，但在若干重要方面偏离了其设计，相关内容将在下文讨论

见第 3.1节。两者在适用范围和功能上也有一些关键差异。TinyRAM 旨在表达任意非确定性计算的正确性，而 MicroRAM 则专注于存在漏洞的程序。例如，C 语言的简洁非交互式知识论证[10] 方法无法直接在零知识中编码内存安全漏洞的证明。相反，它们会编码这样一种知识：存在一条完整、具体的漏洞执行轨迹，其中包含所有局部变量在各时刻的精确值副本，以及执行轨迹中每个时刻的内存值；并且错误必须能从执行的返回值中显现出来。我们的方法直接编码内存漏洞，从而显著缩短了漏洞见证。具体而言，我们可以忽略发现错误后的执行轨迹，也不依赖程序的返回值。

此外，TinyRAM 方法无法扩展到对实际程序中的漏洞进行证明，而且仅在包含少于 1,200 条底层指令的程序上进行过评估 [10]。相比之下，本文提出的优化使我们能够支持包含超过 340,000 行底层代码的程序（表 1 ）。除了可扩展性之外，Micro-RAM 还支持 C 语言中范围更广的子集，包括标准 C 库的大部分内容。

Pantry 和 Buffet [17, 52] 将计算表示为算术约束；满足这些约束的解就是该计算的有效执行轨迹。在实现 TinyRAM 的内存一致性方法后，他们报告的结果比 TinyRAM 好几个数量级。Buffet 支持 C 语言的所有特性，但 `goto` 语句和函数指针除外。为了将计算转换为约束系统，Pantry 和 Buffet 必须将循环展开至公开披露的边界（尽管原始工作没有明确讨论如何编码递归函数，我们推测它们会采用类似方式编码，即使用有界函数内联）。约束系统必须包含条件语句的每个分支以及每个循环的每次迭代（嵌套循环时数量会相乘），这可能导致约束系统规模膨胀；不过，作者指出，这种情况只会出现在退化情形中，在实践中并不常见。Pantry/Buffet 的一种变体采用零知识技术，在保持相同效率优势的同时隐藏状态。在介绍我们的方法时，我们将其所揭示的有关私有输入的信息，与公开循环边界所揭示的信息进行比较（第 3.2.3 节）。

vRAM [53] 通过巧妙的通用预处理进一步提升了效率，使各方能够使用更小的电路，专门用于验证所选输入对应的特定程序。遗憾的是，这种定制电路可能会泄露有关所提供输入的大量信息。我们的公开程序计数器优化（第 4.1 节）试图在定制电路带来的收益与证明者的隐私要求之间取得平衡。

8 结论

随着零知识协议的开发持续取得成功，近期技术已接近能够证明对现实漏洞的掌握，并证明底层代码中细微漏洞利用程序的临界点。本文介绍了如何将编译器设计中的一系列核心技术——即保守指令剖析和欠近似信息流污点标记——应用于优化型证明语句生成器，从而为商用软件生成漏洞证明；而这些漏洞只有在耗费大量时间和空间的情况下才能触发。

我们的实践经验已经在 FFmpeg 中实现了内存不安全的零知识证明，并在 OpenSSL 中实现了信息泄露的证明；后者直接将 Heartbleed 漏洞利用作为见证，证明了如今为关键应用软件中的漏洞构造零知识证明已切实可行。

可用性与伦理考量

我们正在准备将 CHEESECLOTH 的实现开源，以供发表和工件评估。CHEESECLOTH 通过生成漏洞存在性的零知识证明来帮助实现负责任披露，同时对漏洞及漏洞利用程序保密。我们评估中使用的所有漏洞此前均已公开披露，且修复程序已得到广泛部署。因此，本文所述工作不构成对可能有害信息的不道德披露。

致谢

本材料基于美国国防高级研究计划局（DARPA）根据合同编号 HR001120C0085 资助的工作。本文材料中表达的任何观点、研究结果、结论或建议均为作者所有，不一定反映美国国防高级研究计划局（DARPA）的观点。已获准公开发布，不限分发。

参考资料

[1]CVE-2013-0864。https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0864。访问日期：2022-10-10。[2] FFmpeg。https://ffmpeg.org/。访问日期：2022-09- 01。[3] OpenSSL：密码学和 SSL/TLS 工具包。https://openssl.org/。访问日期：2022-09-05。

[4] Picolibc：适用于更小型嵌入式系统的 C 语言库。https://keithp.com/picolibc/。访问日期：2022- 10-10。

[5] Heartbleed 错误。https://heartbleed.com/。访问日期：2022-09-05。

[6]zkInterface：SIEVE 中间表示（IR）提案。https://hackmd.io/@danib31/BkP9HBp2L。访问日期：2022-10-10。

[7] Alfred V Aho、Monica S Lam、Ravi Sethi 和 Jeffrey D Ullman。《编译器：原理、技术与工具》。Pearson Education India，2007年。

[8] Scott Ames、Carmit Hazay、Yuval Ishai 和 Muthura- makrishnan Venkitasubramaniam。Ligero：无需可信设置的轻量级次线性论证。载于 Bha- vani M. Thuraisingham、David Evans、Tal Malkin 和 Dongyan Xu 编，《ACM CCS 2017》，第 2087–2104 页，达拉斯，德克萨斯州，美国，2017 年 10 月 31 日至 11 月 2 日。ACM Press。

[9] Carsten Baum、Alex J. Malozemoff、Marc B. Rosen 和 Peter Scholl。《Mac'n'cheese：具有嵌套析取的布尔电路与算术电路的零知识证明》。载于 Malkin 和 Peikert [37]，第 92–122 页。

[10] Eli Ben-Sasson、Alessandro Chiesa、Daniel Genkin、Eran Tromer 和 Madars Virza。《C 语言的简洁非交互式知识论证：简洁地验证程序执行并实现零知识》。载于 Ran Canetti 和 Juan A. Garay 编，《CRYPTO 2013》第二部分，计算机科学讲义第 8043 卷，第 90–108 页，美国加利福尼亚州圣巴巴拉，2013年8月18–22日。Springer，德国海德堡。

[11] Eli Ben-Sasson、Alessandro Chiesa、Daniel Genkin、Eran Tromer 和 Madars Virza。《TinyRAM 架构规范，v0.991》。https://www.scipr-lab.org/doc/TinyRAM-spec-0.991.pdf，2013年。

[12] Eli Ben-Sasson、Alessandro Chiesa、Eran Tromer 和 Madars Virza。冯·诺依曼架构的简洁非交互式零知识。在第 23 届 USENIX 安全性研讨会（USENIX Security 14）论文集，第 781–796 页，2014年。

[13] Nick Benton。用于静态分析和程序转换的简单关系正确性证明。《ACM SIGPLAN 通讯》，39(1)：14–25，2004 年。

[14] Alexander R. Block、Justin Holmgren、Alon Rosen、Ron D. Rothblum 和 Pratik Soni。《具有（几乎）最小时间和空间开销的公币零知识论证》。载于 Rafael Pass 和 Krzysztof Pietrzak 编，《TCC 2020，第二部分》，计算机科学讲义第 12551 卷，第 168–197 页，美国北卡罗来纳州达勒姆，2020 年 11 月 16–19 日。Springer，德国海德堡。

[15] Alexander R. Block、Justin Holmgren、Alon Rosen、Ron D. Rothblum 和 Pratik Soni。《基于未知阶群、兼顾时间与空间效率的论证》。载于 Malkin 和 Peikert [37], 第 123–152 页。[16] Jonathan Bootle、Andrea Cerulli、Jens Groth、Sune K. Jakobsen 和 Mary Maller。《Arya：用于正确执行程序的近线性时间零知识证明》。载于 Thomas Peyrin 和 Steven Galbraith 编，ASIACRYPT 2018，第一部分，计算机科学讲义第 11272 卷，计算机科学讲义，第 595–626 页，澳大利亚昆士兰州布里斯班，2018 年 12 月 2–6 日。Springer，德国海德堡。

[17] Benjamin Braun、Ariel J Feldman、Zuocheng Ren、Srinath Setty、Andrew J Blumberg 和 Michael Walfish。带状态的计算验证。在第二十四届 ACM 操作系统原理研讨会论文集，第 341–357 页，2013年。

[18] Michael R Clarkson 和 Fred B Schneider。超性质。《计算机安全期刊》，18(6)：1157–1210，2010年。

[19] Dorothy E Denning。《安全信息流的格模型》。ACM 通讯，19(5)：236–243，1976年。

[20] William Enck、Peter Gilbert、Seungyeop Han、Vasant Tendulkar、Byung-Gon Chun、Landon P Cox、Jaeyeon Jung、Patrick McDaniel 和 Anmol N Sheth。《Taintdroid：用于智能手机实时隐私监控的信息流跟踪系统》。《ACM计算机系统汇刊》(TOCS)，32(2)：1–29，2014年。

[21] Zhiyong Fang、David Darais、Joseph P Near 和 Yu-peng Zhang。《零知识静态程序分析》。载于《2021年 ACM SIGSAC 计算机与通信安全会议论文集》，第 2951–2967 页，2021年。

[22] Nicholas Franzese、Jonathan Katz、Steve Lu、Rafail Ostrovsky、Xiao Wang 和 Chenkai Weng。《面向 RAM 程序的恒定开销零知识》。载于 Giovanni Vigna 和 Elaine Shi 编，ACM CCS 2021，第 178–191 页，线上会议，大韩民国，2021年11月15日至19日。ACM Press。

[23] Galois, Inc. Swanky：用于安全计算的一套 Rust 库。https://github.com/GaloisInc/swanky，2019年。

[24] Rosario Gennaro、Craig Gentry、Bryan Parno 和 Mariana Raykova。二次跨度程序与无需概率可检验证明的简洁非交互式零知识证明。载于 Thomas Johansson 和 Phong Q. Nguyen 编，EUROCRYPT 2013，计算机科学讲义第 7881 卷，第 626–645 页，雅典，希腊，2013年5月26日至30日。Springer，海德堡，德国。

[25] Joseph A Goguen 和 José Meseguer。安全策略与安全模型。载于1982年IEEE安全与隐私研讨会，第 11–11 页。IEEE，1982年。

[26] Oded Goldreich、Silvio Micali 和 Avi Wigderson。仅揭示自身有效性，或适用于所有 np 语言的零知识证明系统。《美国计算机协会期刊》（JACM），38(3):690–728，1991年。

[27] Matthew Green、Mathias Hall-Andersen、Eric Hennenfent、Gabriel Kaptchuk、Benjamin Perez 和 Gijs Van Laer。面向现实世界处理器的软件可利用性高效证明。密码学电子印本档案，2022年。

[28]Jens Groth。关于基于配对的非交互式论证的大小。载于 Marc Fischlin 和 Jean-Sébastien Coron 编，EUROCRYPT 2016，第二部分，计算机科学讲义第 9666 卷，第 305–326 页，奥地利维也纳，2016年5月8日至12日。Springer，德国海德堡。

[29] David Heath 和 Vladimir Kolesnikov。一款 2.1 kHz 零知识处理器，采用 BubbleRAM。载于《2020年 ACM SIGSAC 计算机与通信安全会议论文集》，第 2055–2074 页，2020年。

[30] David Heath 和 Vladimir Kolesnikov。用于析取零知识证明的堆叠式混淆。在密码学理论与应用年度国际会议论文集，第 569–598 页。Springer，2020年。

[31]David Heath、Yibin Yang、David Devecsery 和 Vladimir Kolesnikov。人人皆可使用的万物零知识：面向 ANSI C程序、采用缓存式ORAM的快速零知识处理器。载于 2021年IEEE安全与隐私研讨会论文集，第 1538–1556 页，旧金山，加利福尼亚州，美国，2021年5月24日至27日。IEEE计算机学会出版社。

[32] Zhangxiang Hu、Payman Mohassel 和 Mike Rosulek。以次线性摊销成本高效证明非代数语句。载于 Rosario Gennaro 和 Matthew J. B. Robshaw 编，CRYPTO 2015，第二部分，计算机科学讲义第 9216 卷，第 150–169 页，圣巴巴拉，加利福尼亚州，美国，2015年8月16日至20日。Springer，海德堡，德国。

[33] Yuval Ishai、Eyal Kushilevitz、Rafail Ostrovsky 和 Amit Sahai。基于安全多方计算的零知识。载于《第39届ACM计算理论年度研讨会论文集》，第21–30页，2007年。

[34] JasperVijn. GRIT：GBA 光栅图像转换器。https://github.com/devkitPro/grit，2022年。

[35] Neil D Jones、Carsten K Gomard 和 Peter Sestoft。《部分求值与自动程序生成》。Peter Sestoft，1993年。[36] Jonathan Katz、Vladimir Kolesnikov 和 Xiao Wang。《改进的非交互式零知识及其在后量子签名中的应用》。载于《2018年 ACM SIGSAC 计算机与通信安全会议论文集》，第 525–537 页，2018年。[37] Tal Malkin 和 Chris Peikert（编）。《CRYPTO 2021》第 IV 部分，计算机科学讲义第 12828 卷，线上会议，2021年 8 月 16–20 日。Springer，德国海德堡。

[38] Payman Mohassel、Mike Rosulek 和 Alessandra Scafuro。面向 RAM 程序的次线性零知识论证。载于 Jean-Sébastien Coron 和 Jes- per Buus Nielsen 编，EUROCRYPT 2017，第 I 部分，计算机科学讲义第10210卷，第501–531页，法国巴黎，2017年4月30日至5月4日。Springer，德国海德堡。

[39] Andrew C Myers。Jflow：实用的基本静态信息流控制。载于《第26届ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集》，第228–241页，1999年。

[40] Andrew C Myers 和 Barbara Liskov。《信息流控制的去中心化模型》。ACM SIGOPS 操作系统评论，31(5):129–142，1997年。

[41] Nicholas Nethercote 和 Julian Seward。《Valgrind：重量级动态二进制插桩框架》。载于第28届 ACM SIGPLAN 编程语言设计与实现会议论文集，PLDI '07，第89–100页，纽约，纽约州，美国，2007年。计算机协会。

[42] James Parker、Niki Vazou 和 Michael Hicks。《Lweb：多层 Web 应用的信息流安全》。ACM编程语言学报，3(编程语言原理)，2019年1月。

[43] Bryan Parno、Jon Howell、Craig Gentry 和 Mariana Raykova。《Pinocchio：近乎实用的可验证计算》。载于2013年 IEEE 安全与隐私研讨会，第238–252页，伯克利，加利福尼亚州，美国，2013年5月19–22日。IEEE计算机学会出版社。

[44] John C Reynolds. 分离逻辑：一种用于共享可变数据结构的逻辑。载于第17届IEEE计算机科学逻辑年度研讨会论文集，第55–74页。IEEE，2002年。

[45] Andrei Sabelfeld 和 Andrew C Myers. 基于语言的信息流安全性。《IEEE通信领域精选期刊》，21(1):5–19,2003。

[46] Mooly Sagiv、Thomas Reps 和 Reinhard Wilhelm. 基于三值逻辑的参数化形状分析。ACM《编程语言与系统汇刊》(TOPLAS)，24(3):217–298, 2002。

[47] Marc Shapiro 和 Susan Horwitz。快速且准确的流不敏感的指向分析。载于第 24 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集，第 1–14 页，1997 年。[48] Bjarne Steensgaard。近乎线性时间的指向分析。载于第 23 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集，第 32–41 页，1996 年。[49] Deian Stefan、Alejandro Russo、John C Mitchell 和 David Mazières。Haskell 中灵活的动态信息流控制。载于第 4 届 ACM Haskell 研讨会论文集，第 95–106 页，2011 年。[50] G Edward Suh、Jae W Lee、David Zhang 和 Srinivas Devadas。通过动态信息流跟踪实现安全程序执行。ACM SIGPLAN 通讯，39(11)：85–96，2004 年。[51] Riad S. Wahby、Srinath T. V. Setty、Zuocheng Ren、Andrew J. Blumberg 和 Michael Walfish。可验证外包计算中的高效随机存取存储器与控制流。载于 NDSS 2015，圣迭戈，加利福尼亚州，美国，2015 年 2 月 8–11 日。互联网协会。[52] Riad S Wahby、Srinath TV Setty、Zuocheng Ren、Andrew J Blumberg 和 Michael Walfish。可验证外包计算中的高效随机存取存储器与控制流。载于 NDSS，2015 年。[53]Yupeng Zhang、Daniel Genkin、Jonathan Katz、Dimitrios Papadopoulos 和 Charalampos Papamanthou。vRAM：通过与程序无关的预处理实现更快速的可验证随机存取存储器。载于 2018 年 IEEE 安全与隐私研讨会（SP），第 908–925 页。IEEE，2018 年。