Cheesecloth: Zero-Knowledge Proofs of Real-World Vulnerabilities

Santiago Cuéllar* Galois, Inc.
Bill Harris Galois, Inc.
James Parker Galois, Inc.
Stuart Pernsteiner Galois, Inc.
Eran Tromer Columbia University

Abstract

Currently, when a security analyst discovers a vulnerability in critical software system, they must navigate a fraught dilemma: immediately disclosing the vulnerability to the public could harm the system's users; whereas disclosing the vulnerability only to the software's vendor lets the vendor disregard or deprioritize the security risk, to the detriment of unwittingly-affected users.

A compelling recent line of work aims to resolve this by using Zero Knowledge (ZK) protocols that let analysts prove that they know a vulnerability in a program, without revealing the details of the vulnerability or the inputs that exploit it. In principle, this could be achieved by generic ZK techniques. In practice, ZK vulnerability proofs to date have been restricted in scope and expressibility, due to challenges related to generating proof statements that model real-world software at scale and to directly formulating violated properties.

This paper presents CHEESECLOTH, a novel proof-statement compiler, which proves practical vulnerabilities in ZK by soundly-but-aggressively preprocessing programs on public inputs, selectively revealing information about executed control segments, and formalizing information leakage using a novel storage-labeling scheme. CHEESECLOTH's practicality is demonstrated by generating ZK proofs of well-known vulnerabilities in (previous versions of) critical software, including the Heartbleed information leakage in OpenSSL and a memory vulnerability in the FFmpeg graphics framework.

1 Introduction

Ideally, programs that process sensitive information would always execute safely and securely. With this ideal remaining difficult to achieve for the foreseeable future, it is critical that that when programs are found to be vulnerable, the program's affected users are alerted quickly and safely. This requirement presents a challenge: convincingly disclosing a vulnerability appears to require sharing the vulnerability's details (such as an exploit that triggers it), thereby placing users at greater risk in the short term.

A promising approach to disclosing vulnerabilities convincingly yet safely is to leverage *Zero-Knowledge (ZK) proofs*: protocols in which one party—designated as the *prover*—convinces another party—designated as the *verifier*—of the validity of a claim without revealing any additional information about the claim's evidence.

Such a use of ZK proofs has arguably been a conceptual possibility ever since the initial fundamental results establishing that they exist for all problems in NP [26]. It has become more realistic with improvements to underlying ZK protocols and with the emergence of schemes for encoding knowledge of executions of programs written in convenient languages (starting with [10, 24] and discussed further below).

In order to prove vulnerabilities in ZK about practical software, several open problems remaing to be addressed. First, proof frameworks must scale to compile proofs of vulnerabilities that require considerably more steps of execution and space. TinyRAM [10–12] is sufficiently flexible to validate the executions of applications, but it is expensive, in part due to the fact that it simulates every instruction in the modeled CPU's ISA in each step. TinyRAM's performance is surpassed by those of Pantry [17] and Buffet [52], but both frameworks require loops to be unrolled to a public bound: publicly revealing these bounds leaks information about the underlying vulnerability.

A second open problem is to efficiently compile statements from an understandable form. One immediate approach is to execute a program under a dynamic safety monitor for well-understood safety properties, such as those those implemented in Valgrind [41]; however, directly encoding the additional monitoring would induce prohibitively large overhead. Approaches for verifying low-level exploits in ZK [27] rely on being able to efficiently compile directly-understandable properties into statements of control-location reachability.

*Authors listed alphabetically.

To address these problems, we present CHEESECLOTH, an optimizing ZK proof-statement generator that efficently encodes vulnerabilities in practical software. The contributions behind CHEESECLOTH's design include:

1. Optimizations of ZK statements that verify the executions of programs, taking advantage of program structure but without revealing additional information about the execution. Specifically, *Public-PC segments* construct execution traces from segments with public program counters, thus enabling aggressive constant folding, without leaking information about the overall execution trace. Similarly, instructions which are publicly-determined to be executed infrequently are *sparsely* supported (i.e. can't be executed at every step), making the statement smaller.
2. Novel, efficient ZK encodings of memory errors prevalent in practical software, specifically *out-of-bound access*, *use-after-free*, *free-after-free*, and *uninitialized access*. Previous related work focused primarily on proving knowledge of a valid execution without proving existence of a vulnerability [10,11] or encoded proofs of vulnerability using a less efficient memory model [29].
3. A novel, efficient encoding of statements that a program *always* leaks data (when given an exploit as a secret input). Our scheme enables proofs of program properties that are related to, but critically distinct from, existing program monitors and type systems that prove that a program *may* leak data [20,39], optionally in ZK [21].

We implemented these optimizations and encodings in CHEESECLOTH, a full compilation toolchain for encoding vulnerabilities of real-world programs into efficient ZK proofs. The toolchain extends previous approaches based on TinyRAM, and includes a full definition of a novel TinyRAM extension (named MicroRAM) and a compiler to MicroRAM from the LLVM intermediate language, enabling proofs of vulnerabilities in programs provided in C, C++, or Rust.

We evaluated our implementation by proving in ZK the existence of three vulnerabilities in practical systems software. Specifically, we proved that previous versions of the GRIT and FFmpeg [2] graphics processing libraries contained buffer-overflow vulnerabilites, and that the OpenSSL cryptograhy toolkit [3] was vulnerable to the notorious Heartbleed vulnerability [5]. CHEESECLOTH takes the software C/C++ source code and a flag denoting a vulnerabilities class; it combines these with an emulation of the runtime environment (operating system and libraries), and applies the aforementioned techniques, to derive a statement directly provable in ZK. The ZK proof can then be given, as a witness, the concrete exploit used to demonstrate the original attack. CHEESECLOTH contains implementations of powerful program analyses that, when combined with manual program partitioning in some cases, dramatically increase the scale of programs that it can process, compared to a more naive compiler.

The remainder of this paper is organized as follows: Sec. 2 reviews the background that this work builds upon. Sec. 3 presents the implementation details of our CHEESECLOTH compilation pipeline; Sec. 4 covers the critical and aggressive optimizations we make to verify the ZK execution of a program; Sec. 5 describes our ZK encodings to efficiently detect memory and information leakage vulnerabilities; Sec. 6 describes our practical experience using CHEESECLOTH to prove vulnerabilities; Sec. 7 compares our approach to related work and Sec. 8 concludes.

2 Background

In this section, we review prior work on which our contribution builds upon, specifically *Zero-Knowledge (ZK) proofs* of program executions (Sec. 2.1), information leakage by programs (Sec. 2.2), and partial program evaluation (Sec. 2.3).

2.1 Zero-Knowledge Proofs

Zero-knowledge proofs enable a *prover* party to prove to a *verifier* party that the prover knows the correctness of a computational statement (e.g., that a given Boolean circuit is satisfiable), without revealing information about their evidence for the claim (e.g., the witness that satisfied the circuit). There exist ZK protocols for proving knowledge of solutions to all problems in NP [26], and in recent years, numerous efficient protocols have been developed and implemented for ZK proofs of general statements (e.g., [8, 10, 12, 14–16, 22, 24, 28, 29, 31, 32, 38, 43, 51]).

Some of these works specifically address statements about correct execution of programs running on a general-purpose architecture that include Random Access Memory (RAM), where the program is expressed in low-level machine code or a high-level language [9,10,12,14,16,17,22,27,29,31,32, 38,51,52].

Our compiler uses a hybrid of step-by-step CPU emulation, similar to TinyRAM [10–12], a MIPS-like CPU that can simulate programs in C and similar low-level languages that access RAM. The TinyRAM encoder, given a public TinyRAM program and bound on the number of steps of execution to simulate and a private program input, generates an R1CS that is satisfied by encodings of the input. The constraint system consists of **(1)** a family of constraint systems that validate computations purely over registers in each step and **(2)** a novel memory-checking sub-circuit that verifies the correctness of RAM operations using a permutation network. This CPU-unrolling technique is excellent for supporting language features such as data-dependent loops, control-flow and self-modifying code. The technique can also naturally leverage existing tools such as compilers front-ends and libraries.

We combine TinyRAM-style emulation with direct compilation of program blocks into circuit gates [17, 24, 52] (Sec. 4). The compiler's output is a circuit whose satisfiability is equivalent to the existence of a vulnerability in the source program, and whose structure does not reveal the vulnerability or how it may be triggered.

In our evaluation, the underlying ZK protocol is the Mac'n'Cheese [9] protocol for proving circuit satisfiability, as implemented by the Swanky [23] library. This is an interactive protocol, where the prover and verifier engage in multiple rounds of communication to evaluate the circuit, at the end of which the verifier learns that the circuit accepted the secret witness provided by the prover (and nothing else).

2.2 Information Flow

One core contribution of our work is a practical scheme for proving in zero knowledge that a program leaks data, which we have applied to prove that previous versions of OpenSSL leak private data, as triggered by the Heartbleed vulnerability (described in Sec. 5.2 and Sec. 6.3). The scheme's design requires a formal treatment of information flow: specifically, a treatment sufficiently formal that we could generate logical circuits that would be satisfied only by witnesses to leakage. In the interest of space and clarity, we will omit a definition of information flow and leakage for a full programming language, but we will describe ours in sfufficient datail to communicate the key challenges and approaches.

A *labeling* L is a subset of a program's input variables I designated as the *private inputs*, and a subset of its output variables O denoted as *public outputs*. Program P satisfies *noninterference* with respect to L if each pair of inputs that are only distinct at private inputs result in values that are the same at all public outputs; P *leaks* with respect to L if, with respect to L, it does not satisfy noninterference. It follows from the above definition that a leak is witnessed by a *pair* of executions that differ only at L-labeled inputs and produce distinct L-labeled outputs.

Noninterference has a precise but accessible formal definition that can capture the flow requirements of some critical software [20], but its shortcomings in practice are well known [39, 40, 45]: the complete information flow specifications of practical programs often are not noninterference properties, intuitively because programs that take sensitive inputs typically do need to reveal some partial information about them; and even when desired flow properties are noninterference properties, proving that a program satisfies the property in general can involve careful reasoning about unbounded data and control. A rich body of prior work [13, 18, 19, 25] has considered generalizations of noninterference involving equivalences over observable events, along with rich programming languages and type systems and attempt to prove their satisfaction. However, noninterference properties still constitute aspects of a program's complete information flow requirements that unfortunately are both critical and are violated in practice (Heartbleed being a prominent example). This pattern justifies the current work's primary focus on proving noninterference violations.

2.2.1 Labeled Programs and Executions

In their most general form, information flow and leakage are defined over pairs of executions. Practical program monitors [20, 49] and type systems [39] prove facts about all execution pairs, by *labeling* the program's data and control structures with metadata which is tracked through the execution. These approaches can be carried out by a programmer or automated analysis that directly annotates the program or execution. However, the requisite guarantees are different in our proof-of-vulnerability context compared to their usual applications, as seen next.

At a high level, the guarantees provided by dynamic information flow monitors are as follows. A labeling of a program execution over n steps is an assignment from each program variable and step $0 \leq i < n$ to a sensitivity label. A labeling *over-approximates* information flow if, from any two executions starting from states that only differ at high inputs, the program produces results that differ only at high-labeled timestamped storage cells (static analyses and type systems lift this property to be defined over *all* pairs of executions that differ only at sensitive inputs).

Such over-approximation allows for "false positives" in identifying information flows. For example, in a context where only input x is sensitive and the return value is public, the following function `always_true` does not leak any information about its input `secret` because it returns `true` for each input value:

```
bool always_true(bool secret) {
  if (secret) return secret;
  else return !secret; }
```

However, many natural taint analyses would label the returned values as sensitive because it is computed from `secret`.

Over-approximation of potential leaks is often still valuable for aiding programmers to ensure that their program does not leak: falsely determining that a secure program may leak may constitute a nuisance, and may need be mitigated to ensure practicality, but can to some degree to tolerated.

However, in our setting of proving a leak in ZK, it is unacceptable for the verifier to learn only that a program *may* leak. The whole point is to prove that it *does* leak (given the purported exploit). We will thus create a labeling which is an *under-approximation*, i.e., when the labels say so, a leakage *is* present. It will then remain to empirically show that labeling indeed detects leakage for the vulnerabilities of interest.

2.3 Partial Evaluation

In many practical contexts, a program may receive different subsets of its input at different times after it has been written and compiled: e.g., after being installed, a configuration file may be included that remains the same over all executions on distinct inputs subsequently received from a network. A natural objective is, given a program and a subset of its inputs that can be fixed, to generate a specialized program that processes the *remaining* inputs with improved performance. Stated more precisely, for program $P(X,Y)$ with input variables X and Y, a *partial evaluation* of P on an assignment $A : X \rightarrow \mathsf{Words}$ from X to data values Words is a program P_A such that $P(A,B)$ is the same as $P_A(B)$ for each assignment $B : Y \rightarrow \mathsf{Words}$.

Partially evaluating programs in a practical language brings several complexities [35]; the underlying technique amounts to: **(1)** evaluate the program under a symbolic state, in which registers and memory addresses may be mapped either to memory addresses or terms defined over symbolic variables that denoted unknown values; **(2)** using computed symbolic states that describe all possible states at each control point, simplify the control structure at each point. Variations of this technique may be viewed as aggressive extensions and generalizations of the constant propagation analysis and constant folding transformation implemented in conventional optmizing compilers [7].

3 CHEESECLOTH Implementation

CHEESECLOTH produces ZK proofs of real world vulnerabilities. It takes as input a public LLVM program (typically compiled from C, C++, or Rust) and, when run as the prover, a secret exploit that triggers a vulnerability in the program. CHEESECLOTH outputs a ZK circuit that verifies the execution trace of the program and checks whether or not a vulnerability occurred during that execution. The pipeline enables a prover to demonstrate to a verifier that there is a vulnerability in a program while keeping the vulnerability and triggering exploit secret.

CHEESECLOTH produces ZK circuits in multiple standard representations including *Rank-1 Constraints Systems (R1CS)* [10] and SIEVE IR [6]. Because the circuits are serialized in standardized formats, CHEESECLOTH is agnostic to the ZK protocol applied. When run as the prover, CHEESECLOTH outputs the accompanying witness for the circuit.

CHEESECLOTH can be extended to check different properties about a program's execution. Users can selectively enable which extensions to run by providing different input flags to the compilation pipeline. These extensions are how the memory and information leakage vulnerability detection checks described in Sec. 5 are implemented. This section covers the baseline design of the CHEESECLOTH compilation pipeline which includes (1) the MicroRAM assembly language, (2) the MicroRAM Compiler, and (3) the Witness Checker Generator. Sec. 4 describes optimizations for this design that enable it to scale to real world vulnerabilities.

3.1 MicroRAM

The MicroRAM assembly language is critical to CHEESECLOTH. It is the core IR language that CHEESECLOTH operates on and is the language that the MicroRAM Compiler compiles LLVM programs to. The Witness Checker Generator produces ZK circuits that verify program executions according to MicroRAM's architecture.

MicroRAM is heavily inspired by TinyRAM [10, 11], which is a practical and efficient assembly language with a simple transition function that is ideal for ZK execution verification. We describe MicroRAM and its architecture below, and we precisely describe how its design diverges from TinyRAM in Sec. 3.1.1.

MicroRAM is a random-access machine designed to efficiently detect vulnerabilities in program executions. It is a reduced instruction set computer (RISC) with a Harvard architecture and byte-addressable random-access memory.

MicroRAM instructions are relatively simple and include 4 boolean operations, 8 arithmetic operations for signed and unsigned integers, 2 shift operations, 5 compare operations, 2 move operations, 3 jump instructions, 2 operations for reading and writing to memory, and 1 answer operation that returns and halts the execution. Floating-point and vector arithmetic are not directly supported in the MicroRAM machine and must be implemented in software. Instructions take two registers and one operand (either a register or an immediate) as arguments. As an example, instruction `xor ri rj 255` writes to register `ri` the exclusive-or of register `rj` and the immediate `255`. CHEESECLOTH extensions like those described in Sec. 5 can introduce additional instructions as needed.

The state of the MicroRAM machine consists of the program counter (`pc`), k 64-bit registers, a memory of 2^{64} 64-bit words, a flag indicating whether or not the execution so far is valid (`inv_flag`), and a flag tracking whether a vulnerability has occurred (`vuln_flag`). CHEESECLOTH extensions can extend the state of the MicroRAM machine as well.

To demonstrate the existence of a vulnerability in a program, a prover must present a secret input that results in a valid execution trace that triggers a vulnerability. Formally, given a MicroRAM program, P, and an initial memory, m_0, $P(m_0)$ demonstrates a vulnerability in T steps if `inv_flag` is `false` and `vuln_flag` is `true` in the final MicroRAM state of the program's execution trace. `inv_flag` is set to `false` if any of the checks validating the program's execution fails. The extensions implementing the vulnerability detection checks set `vuln_flag` to `true` if they observe a vulnerability during the program's execution.

3.1.1 Beyond TinyRAM

As mentioned above, our MicroRAM machine is inspired by TinyRAM. Here we report on how MicroRAM's design departs from the TinyRAM model.

- MicroRAM's memory model is byte-addressable while TinyRAM is word-addressable. Byte-addressable memory is necessary to support functionality like string manipulations and packed structs, without adding subroutines to access bytes within full words.
- TinyRAM receives input via input tapes. In MicroRAM, input is passed directly in memory, which saves many cycles that TinyRAM spends copying input to memory. A MicroRAM program can request non-deterministic advice in several ways, however the prover does not have to commit to the advice ahead of time on a tape; instead they provide the advice upon request. This approach is better suited to support backends that exploit parallelism or streaming, and it results in smaller circuits.
- TinyRAM uses a 1-bit condition flag for branching while MicroRAM does not. This is advantageous since MicroRAM targets a variety of backends including non-boolean arithmetic circuits where the flag is more expensive than a regular register[1]. In addition, the semantics without a flag are much simpler so the compiler, interpreter, and circuit generator are simpler as well. We found that even when targeting boolean circuits, the benefits of having a condition flag are outweighed by the extra complexity.
- We have not yet explored using a von Neumann architecture [12] for MicroRAM because, despite the asymptotic benefits, the instruction fetching circuit is not yet a limiting factor in our ZK statements.

3.2 MicroRAM Compiler

The MicroRAM Compiler is implemented as a LLVM backend that takes LLVM IR programs as input and produces MicroRAM assembly as output. We currently support C, C++, and Rust programs by compiling them to LLVM IR with the *Clang* and *rustc* compiler frontends. Support for other languages such as C#, Haskell, or Scala can be added in the future by connecting their appropriate LLVM frontends and writing the appropriate standard libraries.

Our compiler backend supports a large subset of the LLVM IR language. The compiler supports all boolean and arithmetic operations for integers of different sizes, bitwise operations, all non-concurrent memory operations including pointer arithmetic with `getelementptr`, conversion operations, function calls, variable arguments, comparisons, and `phi` nodes. Complex operations like floating-point operations are implemented in software via a LLVM compiler pass.

Exceptions, and all exception handling instructions, are not supported; but we we can still tolerate programs with exceptions as long as the prover is disclose that the execution of interest, which triggers a vulnerability, does not throw any exceptions. This is since the MicroRAM Compiler translates all exception handling instructions to traps that mark the trace as invalid by setting the `inv_flag` flag. By inserting traps, the MicroRAM Compiler can process programs with any number of unsupported features, as long as the prover is willing to reveal that those features are not involved in the vulnerable execution. With this simple trick, users can compile real-world programs without having to manually remove unsupported features. When enabling traps, provers must take care not to reveal too much information about the underlying vulnerability. Sec. 3.2.3 presents a more detailed discussion about the security implications of how proof statements can reveal information about their witnesses.

3.2.1 Standard library

MicroRAM supports a significant portion of the C standard library and POSIX system calls, using Picolibc [4]: a library that offers the C standard library APIs and was originally designed for small embedded systems with limited RAM. Picolibc supports multiple widely deployed target architectures, including ARM, RISC-V, and x86-64.

We implemented MicroRAM as a target architecture for Picolibc. This enables the MicroRAM compiler to support most of the C standard library and POSIX system calls. It is also convenient as it allows prover to publicly customize the behavior of system calls. For example, in our case study of OpenSSL, the victim server receives the malicious request from the attacker over the network. We customized the behavior of `read` when compiled natively to intercept and record all data received over the network. When compiled for MicroRAM, `read` returns the previously recorded exploit request, which is loaded from secret memory. We also customize the implementation of `malloc` and `free` to efficiently detect memory vulnerabilities (Sec. 5.1.1).

3.2.2 Generating advice

As we will see in later sections, CHEESECLOTH requires nondeterministic advice to efficiently generate a ZK circuit that verifies the consistency of memory in an execution (Sec. 3.3) and the presence of a vulnerability (Sec. 5.1). To aid the prover in producing that advice, the MicroRAM compiler runs two interpreter passes. The first pass executes the program without any advice and records the necessary advice. The second execution runs the nondeterministic semantics and records the trace, which is passed to the Witness Checker Generator to produce the witness for the prover.

[1] If full words fit in a field element, then the flag is the same size as a register, but requires special circuitry and has more restrictions.

3.2.3 Security

MicroRAM produces zero-knowledge proofs which ensure that no *additional information* is revealed about the witness. However, the proof statement itself can reveal information about the secret input. For example, in MicroRAM and TinyRAM the circuit reveals a time bound T on the execution length. In Pantry/Buffet, the circuit discloses an upper bound T_i on every loop (and recursive function) in the execution. In vRAM [53], every instruction run during the execution is revealed to the verifier. We argue that a formalization of this information leakage is necessary. Interesting and important future work will be to define a formal framework to analyse how secure these encodings are.

3.2.4 Preprocessing public inputs

One opportunity for aggressive optimization is to publicly evaluate logic that is determined by the program's public inputs. Many practical programs collect inputs from multiple sources, some of which are not secret (i.e., irrelevant to the vulnerability). If the prover and verifier agree when defining a proof statement that only some inputs are sensitive secrets (e.g., data packets received from a network connection) while others are not (e.g., straightforward configuration options), then the resulting proof statement could be immediately optimized by generating the proof statement and partially evaluating the resulting circuit on its input wires corresponding to non-sensitive inputs.

CHEESECLOTH supports such cases with a compiler pass that determines the largest program prefix in which no operation depends on secret inputs. The MicroRAM compiler then separates the public prefix from the remaining program suffix and compiles them separately. When the interpreter is executed by both the prover and verifier, it executes the prefix and defines a public snapshot of the resulting state, including both registers and memory. When executed by the prover, the interpreter then executes the remaining suffix using both the snapshot and their private input generate to generate the statement's witness. In practice, this simple optimization has significant impact, reducing the number of execution steps in OpenSSL's ZK proof statement from 25M to 1.3M (Sec. 6.3).

The compiler optimization implements a relatively restricted form of partial evaluation and constant folding (Sec. 2.3). Our initial experience indicates that further extensions could improve CHEESECLOTH's performance drastically: a key technical challenge is that while programs may perform much processing of public data over the course of the entire execution, the processing is often interleaved with computation over sensitive inputs. Evaluating each of the interleaved phases of public computation is sound in principle, but can only be automated by ensuring that regions of storage used by public and secret phases are disjoint. Such automation could potentially be achieved by applying points-to and shape analyses [46–48], including separation logic [44].

3.3 Witness Checker Generator

The Witness Checker Generator takes as input a MicroRAM program and generates a ZK circuit, serialized in standardized formats including R1CS and SIEVE IR. It also accepts nondeterministic advice as input and outputs the secret witness to the circuit when run as the prover.

The Witness Checker Generator builds arithmetic circuits for the prime field $2^{128} - 159$. As an optimization, it automatically constant folds gates that are independent of secret inputs. To scale to large circuits and avoid running out of memory, it streams the circuit serialization to a file. This streaming is independent of secret witnesses, so the same circuit is generated for the prover and verifier.

The nondeterministic advice the Witness Checker Generator accepts provides a description of a program's execution together with the advice necessary to run it. Concretely, the advice for an execution of T steps contains the initial program memory, the T MicroRAM states making up the execution trace, and a mapping from step number to additional advice given at each step. This additional advice includes memory ports for what is read or written to memory and stutters that indicate the execution should pause for the current step.

The Witness Checker Generator produces a ZK circuit that verifies that the witness describes a valid execution trace for the program and that a vulnerability occurs. The circuit is split into four key pieces: (1) the transition function circuit, (2) the memory consistency circuit, (3) a state transition network, and (4) public-pc segments. We describe the first two here, which follow a similar structure to the circuit construction for TinyRAM [10]. The other two are described later in Sec. 4.1.

Transition function circuit. The transition function circuit checks a single step of execution. These checks are chained together to validate the entire execution trace. Fig. 1 shows pseudocode for the transition function circuit. It takes as input the circuit's wire representation of the current MicroRAM state, the next state, and any additional advice needed for the current step. The circuit then fetches the instruction to execute based on the program counter and pulls out the instruction's argument values by indexing into machine registers. It calculates the expected result of the step by multiplexing over the instruction. Finally, the circuit ensures that the calculated expected state matches the next state provided as advice.

Memory consistency circuit. The memory consistency circuit is similar to TinyRAM's except addresses are byte-addressable instead of word-addressable. Each step may have a corresponding memory port advice that states the address and what was read or written to memory. The transition function circuit verifies that the execution trace matches the memory port advice. All of the memory ports are sorted by address and step number. The memory consistency cir-

```
fn transition_func(circuit, current_st, next_st) {
  let expected_st = current_st.clone();
  let instr = fetch_instr(circuit, current_st.pc);
  let arg1 = index(circuit, current_st.regs, instr.op1);
  let arg2 = index(circuit, current_st.regs, instr.op2);

  let result = circuit.mux(instr.opcode == XOR,
                 xor(circuit, arg1, arg2), ...);
  expected_st.pc = circuit.mux(
                     is_jump(circuit, instr.opcode),
                     result,
                     circuit.add(current_st.pc, 1));
  write_index(circuit, expected_st.regs, instr.dest,
                result);

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

Figure 1: Pseudocode for the transition function circuit that validates a single MicroRAM step.

cuit linearly scans the memory ports to ensure that all reads and writes to a given address are consistent with the previous memory operation. For example, a read should return the same value that was previously written to an address. Finally, the memory consistency circuit checks that the sorted memory ports are a permutation of the memory ports used by the transition function circuit. Sec. 5.1 describes how these checks are enhanced to efficiently detect memory vulnerabilities.

4 Optimizations

This section describes two of CHEESECLOTH's key optimizations: constructing executions with public program counters (Sec. 4.1) and tuning steps based on instruction sparsity (Sec. 4.2). Sec. 6.4 contains an empirical evaluation of the optimizations' effectiveness.

4.1 Public-PC segments

The MicroRAM machine is design to minimize the size of the circuit that checks the transition function circuit. However, even with MicroRAM's small instruction set, the transition function circuit is still large. This is due to the fact that every transition function must support every operation in every step of execution. What if we could remove *all* the unused functionality? This is the approach of vRAM [53], where the circuit is tuned to check the instruction that is executed at each step. The resulting circuit is much smaller, but unfortunately the trace of executed instructions is revealed. The values in memory and registers would still be kept secret, however a verifier could easily discover where the vulnerable code is in the program. In this section, we present *public-pc segments* which generate much smaller circuits without revealing the trace.

A *public-pc segment* is a sequence of transition circuits with a hardcoded and public program counter. Using constant folding, all the instruction fetches of the public-pc segments are known and the unused functionality of every step can be removed. For example, with a public program counter, the `fetch_instr` and subsequent `mux` operations over the instruction in Fig. 1 can be constant folded away. We generate public-pc segments for all straightline code segments in a program, and we implemented a compiler pass that uses a naive control-flow analysis to estimate how many times each segment will be used. The analysis takes a global bound specifying how many times to unroll loops and estimates how many times a function will be called by counting the number of call sites for that function in the program.

To preserve the security of the trace, the cycle counter of segments is kept private. In addition, we introduce a state routing network so that the end state of a segment could be the initial state for any other segment in the circuit. Just like the memory routing network, the routing information for the state routing network is given by the prover and kept secret.

As a further optimization, we avoid using the state routing network when possible. For example, when a public-pc segment branches to two statically known locations, we directly connect the end state of that segment to the segments representating those two locations.

It is possible that the bound for unrolling loops is not large enough to support certain executions, so the pipeline would not generate enough public-pc segments for a section of code. As backup, the pipeline also produces *private-pc segments* which are just like public-pc segments except the program counter is not revealed. Private-pc segments look similar to a much smaller TinyRAM circuit with the difference that their start and end states come from the state routing network. The circuits for these segments are significantly larger, but can execute any part of the program at any point during execution.

4.2 Sparsity

With the naive CPU unrolling described in Sec. 3.3, every transition function must contain a memory port, which causes the memory consistency network to grow at a rate of $O(T \log T)$, where T is the number of steps executed. Unfortunately, most of those gates are wasted by execution steps that do not access memory. CHEESECLOTH mitigates this excess by removing some of the unused memory ports, thereby reducing the size of the memory consistency circuit.

The key observation for this optimization is that memory operations are rarely contiguous. Even when a program performs a memory-intensive operation, other instructions are often interleaved between memory instructions. For example, when adding the values in a buffer, it takes some steps to increment the pointer and add the values between memory reads. This enables us to share one memory port among s contiguous steps, shrinking the memory consistency network by a factor of s.

We define the *memory sparsity*, s, as the number of steps that share a single memory port. CHEESECLOTH chooses s based on a static analysis of the code. The analysis determines the minimum distance between two memory operations in any possible execution. Across statically-unknown jumps (e.g. calling a function from a pointer dereference), the analysis naively considers all the instructions the control flow can possibly jump to. This memory sparsity number s is then used by the MicroRAM Compiler and Witness Checker Generator to generate the optimized circuit.

Given a memory sparsity s, the Witness Checker Generator will group s consecutive steps and create a single memory port for all of these steps. A multiplexer connects the single memory port to the entire group and sends the result, using nondeterministic advice, to the right step (if any).

If s is larger than the actual sparsity displayed by a trace, then (if unlucky) multiple memory accesses can fall into the same group of steps, which has a single memory port. CHEESECLOTH handles this situation by inserting stutter instructions that delay memory operations until they are pushed into the next group with separate memory ports. Inserting stutter instructions can be expensive, but reducing the size of the memory consistency circuit is more beneficial (Sec. 6.4). In future work, we will explore swapping program instructions to reduce stutter instructions and determine the optimal s parameter for most programs.

5 Encoding Vulnerabilities

This section describes how CHEESECLOTH encodes two prevalent and critical classes of software vulnerabilities: memory unsafety (Sec. 5.1) and data leakage (Sec. 5.2).

5.1 Memory unsafety

We now describe how CHEESECLOTH efficiently models memory and represents memory vulnerabilties. In CHEESECLOTH, memory is an array of 2^{64} bytes with half reserved for the heap and the rest for global variables and the stack. Our approach is to keep track of valid memory (e.g. allocated arrays) and report a vulnerability (i.e., set `bug_flag`) when the program accesses non-valid memory. At the start of the execution, the only valid memory is where the global variables are stored and, during execution, malloc makes allocated regions valid and free makes them invalid again. With this technique we can catch the following memory errors:

- *Uninitialized access*. All uninitialized memory is invalid, so any use triggers a bug.
- *Use-after-free*. When a region is freed it becomes invalid, so any use triggers a bug.
- *Free-after-free*. The implementation of free starts by reading a word from the region to be freed, if the region is not valid it triggers a bug.

```c
void* malloc(size_t size) {
  // Get pointer from advice
  char* addr = __cc_malloc(size);
  /* Compute and validate the size of
   * the allocation provided by the
   * prover. */
  size_t region_size =
    1ull << ((addr >> 58) & 63);
  /* The allocated region must have
   * space for `size` bytes, plus
   * an additional word for metadata.
   */
  __cc_valid_if(
    region_size >= size + sizeof(uintptr_t),
    "allocated region size is too small");
  /* `region_size` is always a power of
   * two and is at least the word size,
   * so the address must be a multiple
   * of the word size. */
  __cc_valid_if(addr % region_size == 0,
    "allocated address is misaligned for"
    "its region size");
  /* Write 1 (allocated) to the metadata
   * field, and poison it to prevent
   * tampering, invalidating the trace
   * if the metadata word is already
   * poisoned (this happens if the
   * prover tries to return the same
   * region for two separate
   * allocations). */
  uintptr_t* metadata = (uintptr_t*)
    (addr + region_size - sizeof(uintptr_t));
  __cc_write_and_poison(metadata, 1);

  // further computation...
  return (void*)addr; }
```

Figure 2: Implementation of non-deterministic `malloc`.

- *Out-of-bound access*. If the program accesses an address out of bounds, that new location *might* (see below) not be valid and this triggers a bug.

It is clear that a normal execution with such bound checking *might* miss out-of-bound access bugs, when the access happens to fall on another valid region, and free-after-free/use-after-free bugs, if an intermediate malloc makes the region valid before the bug is triggered. However, we only need to show that the bug exists in one execution, so we implement a malloc guided by nondeterministic advice; this lets the prover choose the allocation layout to ensure the bug is triggered.

While the techniques described here are specific to heap memory bugs, the same ideas can be applied to the stack.

5.1.1 Encoding dynamic memory allocation

An implementation of malloc with nondeterminism poses its own challenges. If left unchecked, the prover could manufacture an execution that triggers a false bug. For example the prover could malloc overlapping regions such that if one is freed and the other one is accessed a false bug is triggered. Thus, our implementation of malloc and free (Fig. 2) focuses on verifying that the nondeterminisitc choices are legal. If foul play is detected, the execution is flagged as invalid with `inv_flag` and will not be accepted by the verifier.

To ensure that malloc never returns overlapping regions, we predetermine aligned non-overlapping regions of different sizes for malloc to choose from. Concretely, we divide memory into 2^6 pools of size 2^{58}, then subdivide pool i into regions of size 2^i. malloc rounds up the requested size to the next power of two, then returns the start of an unallocated region of that size. For example, malloc(15) must return a region in the 4th pool and be 16-byte aligned. In fact, we can easily verify that malloc has allocated a correct region just by looking at the pointer returned: the first 6 bits determine the pool and the rest the alignment.

Finally, malloc must not return the same pointer twice without it being freed in between. To do so, we add to each region one word reserved for metadata that is marked and made invalid when the region is allocated. If the region was allocated again, the invalid metadata would be made invalid again which makes the trace invalid by setting `inv_flag`.

Furthermore, an implementation of malloc/ free that tracks the validity of all memory locations would be quite inefficient. Luckily, the prover knows exactly where the bug will happen and thus the malloc/ free implementation only needs to track the status of that location. At the beginning of the execution, the prover commits to a secret location stored in the global variable __cc_memory_error_address and then malloc/ free only track the validity of that location. In particular, if an allocated/freed region does not contain __cc_memory_error_address then malloc/ free do not check for errors, and run in constant time.

5.2 Data leakage

A straightforward approach to proving leakage would be to directly encode the definition of noninterference in the ZK circuit. This could be accomplished by verifying two program executions where only sensitive inputs are distinct but public outputs are distinct. However, such an approach would result in a statement of twice the size required for validating a single execution. Instead, we might hope to prove a leak using a single execution in which storage is annotated with labels (Sec. 2.2). However, such systems tranditionally have only been designed to prove that a program *may* leak information, which is unacceptable for definitively proving a leak without providing a violating execution directly (Sec. 2.2.1).

Specifying leakage To identify sensitive sources and sinks, the instructions `source` and `sink` are added to the MicroRAM instruction set, and are directly wrapped by user-level functions `taintSource` and `taintSink`, respectively. `source` annotates that a given byte of data carries sensitive data; `sink` annotates that a given byte is output to a channel. Instantiating the general definitions of information flow and leakage (Sec. 2.2) for this extended ISA, a MicroRAM program leaks if it has two executions whose inputs only differ at addresses given to `source`, but result in different values at an address given in calls to `sink`. Leakage is established by the prover and verifier collaborating to extend the subject program to call the `taintSource` and `taintSink` to annotate sensitive sources and sinks.

Proving leakage To soundly and precisely prove leakage, we propose a novel labeling system that tracks what program storage may and *must* hold secret. There are four labels, denoted and partially ordered as

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

with a least-upper bound (i.e., *join*) operation denoted $\sqcup$. Labels ℓ_0 and ℓ_1 annotates data that *must* belong to one of two principals; $\top$ denotes that the data's sensitivity is unknown; $\bot$ denotes data that *must* not be influenced by a principal. With this labeling scheme, leakage of ℓ-labeled data written to a ℓ_c-labeled sink *must* occur when $\ell \neq \top \land \ell \not\sqsubseteq \ell_c$.

MicroRAM state is extended so that every register and byte of memory is associated with a label, similar to previous leakage monitors [20, 42, 49, 50]. Two additional labels model effects of instructions other than register arithmetic. The *control context* label γ is maintained to be $\bot$ if the program execution has not branched on secret data, and $\top$ otherwise; similarly, the *storage context* label σ is maintained to be $\bot$ if the program has not stored to a secret address, and $\top$ otherwise.

Each assignment `x:=e` sets the label of `x` to $L(e) \sqcup \gamma \sqcup \sigma$ (where $L(e)$ is the label of e, defined below); thus, if the program has branched on secret data or written to a secret address, the label of `x` is set to $\top$. If e is an arithmetic/logical operation $f(y)$, then $L(e)$ is $\bot$ when $L(\mathtt{y})$ is $\bot$ and $\top$ otherwise; $L(e) = L(\mathtt{y})$ if f is a *bijection*: our current implementation conservatively only labels single-register expressions (i.e., copy sources) as $L(\mathtt{y})$. If e is a load `*p`, then $L(e)$ is $L(\mathtt{*p})$. Conditional branches update γ and memory stores update σ according to the labels' descriptions; we omit formal descriptions here, due to space constraints.

Plenty of natural programs leak but cannot be proved to do so by this labeling system, potentially because a leakage happens after branching or storing to a $\top$-labeled value, or because a secret value is propagated over an operation not recognized as a bijection. Such cases restrict the situations in which the labeling scheme can be applied to prove leakage, but do not threaten its validity when it claims that a given program leaks. They might be addressed in future work that refines instruction interpretations using valid logical axioms (e.g., the fact that for each value x, $x + 0 = x$). Such threats and mitigations are dual to threats to precision, and possible refinements when proving that a program execution *may* leak.

6 Evaluation

We evaluate CHEESECLOTH with three case studies that demonstrate ZK proofs of real world software vulnerabilities. The vulnerabilities scale by code size and execution trace length to showcase the capabilities of CHEESECLOTH. We also benchmark the optimizations (Sec. 4) to evaluate their effectiveness.

Tab. 1 presents the results of using CHEESECLOTH to produce ZK proofs for our case studies which include GRIT, FFMPEG, and OpenSSL. For each case study, we report the size of the program in terms of the number of MicroRAM instructions, the number of execution steps required to demonstrate the vulnerability, and the number of multiplication gates in the resulting ZK circuit. We prove satisfiability of the ZK circuit using the Mac'n'Cheese [9] interactive ZK protocol, as implemented by the Swanky [23] library. We record the protocol running time and communication cost between the prover and verifier. All measurements were performed on a 128 core Intel Xeon E7-8867 CPU with 2 TB of RAM running Debian 11, although our implementation typically uses considerably less memory (Tab. 1).

6.1 Memory unsafety in GRIT

The GBA Raster Image Transmogrifier (GRIT) [34] converts bitmap image files to a graphics format that is readable by the Game Boy Advance. A bitmap image includes headers, a palette array indicating the colors in the image, and the pixels for the image. For 24bpp images, GRIT's parser assumes the palette size is zero and allocates a buffer without space for the palette. When populating the buffer, it checks the image header for the number of palette entries without checking that this matches the assumed palette size that was used during allocation. As a result, a malformed 24bpp image can write an arbitrary amount of data (up to the length of the file) past the allocated buffer.

To demonstrate this memory error, we construct a 24bpp exploit image with 0x3000 bytes of pixel data and 12 bytes of palette data. On Linux, the 12 byte overflow overwrites heap metadata and triggers an assertion failure in the memory allocator. When run through CHEESECLOTH, we generate a ZK proof that a memory error is triggered within six thousand steps of GRIT's execution without revealing the triggering image or where the error occurred in the code.

6.2 Memory unsafety in FFmpeg

FFmpeg is a tool for recording, converting, and streaming audio and video [2], and is used in popular software projects such as Chrome, Firefox, iTunes, VLC, and YouTube. FFmpeg is written in C and has been plagued by vulnerabilities that compromise memory safety, enabling attackers to execute code and share local files over the network. Versions of FFmpeg prior to v1.1.2 contained a vulnerability [1] caused by the memory error in the function `gif_copy_img_rect`, which copies the frame of a GIF file between buffers. Previous versions of `gif_copy_img_rect` insecurely calculated a pointer to the end of a memory buffer by directly using the input image's height. This calculation allowed an attacker to provide a carefully crafted GIF which causes FFmpeg to write to memory outside of an array's bounds.

To prove memory unsafety of FFmpeg in ZK, we manually crafted a GIF image that exploits the described memory vulnerability. We passed this image and a program module that invokes FFmpeg's video decoder to CHEESECLOTH, which generated a proof of a out-of-bound access. The only facts revealed about the exploitative GIF are what are implied by the fact that it trigger an out-of-bound access within 76K steps of execution.

Preprocessing FFmpeg on public inputs There was potential to aggressively optimize FFmpeg's proof statement, which was ultimately achieved by applying CHEESECLOTH's constant folding transformation pass after manual program partitioning. The need for partitioning arose due to the interleaving of public and secret computation in the GIF modules, which executes by: **(1)** demultiplexing a given *secret* GIF file into a sequence of data packets; **(2)** initializing the state of the decoder, using *public* configuration settings; **(3)** executing the codec that contains the vulnerability.

Although phase (2) computes entirely over public data, it would not be optimized by CHEESECLOTH's constant folding pass because the pass halts upon detecting computation that uses secret data, and thus would not optimize any program segment after phase (1). To address this issue, we manually partitioned the program by phase, applied CHEESECLOTH's constant folding pass to each, and linked the resulting optimized MicroRAM code. In general, our case study of FFmpeg motivates the further study and design of more aggressive constant folding passes, which might apply more sophisticated static program analyses (Secs. 2.3 and 3.2.4).

6.3 Leakage in OpenSSL

OpenSSL [3] is a widely deployed open-source cryptographic library that contains implementations of the SSL and TLS protocols. OpenSSL versions 1.0.1 to 1.0.1f contained a devastating vulnerability dubbed *Heartbleed*, discovered in 2014 [5], that could be exploited by a remote attacker to completely leak information stored over the protocol's execution, including other clients' sensitive information and private keys.

Comprehensive descriptions of SSL and OpenSSL are beyond the scope of this paper; for the purposes of our work, it suffices to note that SSL parties support multiple requests, including both requests to store data from the another party and requests to reply to a *heartbeat* signal: a signal sent only

Program	Code size (K instrs)	Execution steps (K)	Mult gates (M)	Protocol time	Protocol memory
GRIT	3	5	26.7	3m 40s	845 MB
FFmpeg	24	79	672.7	1h 22m	19 GB
OpenSSL	340	1,300	17,049.5	36h 45m	460 GB

Table 1: Results for generating and running a ZK proof of software vulnerability for each case study.

```
void process_heartbeat(SSLRequest *req) {
  unsigned int len = parse_heartbeat_len(req);
  unsigned char *heartbeat = get_heartbeat(req);
  unsigned char *response = malloc(len);
  memcpy(response, heartbeat, len);
  write(response, len); }
```

Figure 3: Pseudocode depicting the Heartbleed vulnerability.

to obtain a response to ensure that the other party is still responsive.

The heartbeat request and response is critical to the operation of the Heartbleed vulnerability. A well-formed request consists of a *data buffer* d and a *length field* $n < |d|$. A correct response to such a request returns the first n bytes contained in d. However, a party could potentially transmit an *ill*-formed request, in which $n > |d|$. The correct response to such an ill-formed request is to reject it.

The implementation of OpenSSL (illustrated by the pseudocode function `process_heartbeat` in Fig. 3) crucially failed to implement this aspect of the protocol and instead returned the n bytes of memory contiguous with the input buffer. `process_heartbeat` takes a heartbeat request from a client and echos the provided heartbeat string back. It does so by first parsing the length of the heartbeat string from the client's request. The function then gets a pointer to the heartbeat string in the request. Next, it allocates a response buffer and copies `len` bytes from the heartbeat string into the response buffer, which is subsequently sent back to the client. Since `process_heartbeat` does not check the provided heartbeat length against the actual length of the heartbeat string, if the claimed length is larger than the actual length of the provided heartbeat string, memory beyond the client's request is sent back to the client. This is practically exploitable, and has been demonstrated to reveal sensitive in-memory data such as cryptographic keys and passwords.

Using CHEESECLOTH, we proved in ZK that OpenSSL version 1.0.1f leaks arbitrary user information in 1.3M steps of execution, propagating data purely over register copies, loads, and stores; while the statement reveals a bound on the amount of computation required to perform the leak and information about the types of instructions used to perform the leak (described below), it gives no direct indication of what validation is missing in the function for processing heartbeat requests, or that heartbeat requests are involved in the leak at all. We describe the statement proved, along with technical challenges and solutions, in more detail below.

```
int login_handler(
  SSLConn *c, char *password, int len) {
  ...
  label l = getLabel(c);
  for (size_t i = 0; i < len; i++)
    taintSource(password + i, l);
  ... }
int ssl3_write(
  SSLConn *c, char *buf, int len) {
  ...
  label l = getLabel(c);
  for (size_t i = 0; i < len; i++)
    taintSink(buf + i, l);
  ... }
```

Figure 4: Versions of the OpenSSL functions `login_handler` and `ssl3_write` that we augmented with operations that specify information sources and sinks. Passwords are tainted with the label of the current connection, and leaks are detected if data written to the network has a label from a different connection.

Specifying OpenSSL's leakage A primary challenge of our work was to provide a scheme for identifying sensitive sources and sinks such that:

1. A verifier with only an understanding of the data that a subject program handles should be able to inspect the modified program and definitively conclude that it correctly defines information sources and sinks.
2. Any modifications to the program to enable the definition of sources and sinks between which information is leaked must not reveal additional information about the leak's triggering input.

Our mechanism for defining sensitivity of sources and sinks consists of the designated functions `taintSource` and `taintSink` (Sec. 5.2). We found that such a library served well for specifying information flow in in OpenSSL; pseudocode of C functions modified in the OpenSSL codebase to label sources and sinks are given in Fig. 4. The function `login_handler`, given an SSL connection `c` and a buffer `password` presumed to contain `len` bytes of sensitive information to be transmitted over `c`, labels `len` addresses beginning with `password` with the label of `c`. The function `ssl3_write`, given an SSL connection `c` and buffer `buf` presumed to output `len` bytes, denotes sinks at the output channel with the label of `c` for `len` addresses beginning with `buf`.

The modifications to `login_handler` and `ssl3_write`

Program	Mult gates without public-pc segments (M)	Mult gates without sparsity (M)
GRIT	42.3 (37%)	27.9 (4%)
FFmpeg	716.9 (6%)	709.9 (5%)

Table 2: The number of multiplication gates in the circuit with the different optimizations disabled, as well as the percentage increase in size over the baseline numbers from Tab. 1.

illustrate the utility of first-order labels that can be operationally collected and set, as opposed to operations that set addresses as only high sources or low sinks, even in a setting in which the information belonging to only one principal is of interest. By using first-order labels, we we able to write small specification functions that only unified the labels between a network connection and a given buffer, and then succinctly modified the original program logic in contexts that readily provided a connection and related buffer.

Proving OpenSSL's leakage Once OpenSSL has been suitably modified to call the `taintSource` and `taintSink` functions, its leakage can be proved by generating a statement whose solution corresponds to an execution of a server running OpenSSL that leaks sensitive data from one connection to another connection. We have generated such a statement where the server first responds to a public login request where the password is marked as sensitive. The server then handles a secret malicious heartbeat request that returns the password from the previous request's connection.

Using CHEESECLOTH we prove OpenSSL's leakage by validating the previously described execution which is derived from one of its originally disclosed exploits. The leakage is detected through the source and sink annotations according to our proposed must-leak labeling scheme (Sec. 5.2) and the verifier only learns an upper bound on the length of the malicious request. We found that the labeling scheme enabled leakage to be proved much more efficiently, reducing the overall circuit size by 30.6% over the two trace approach. CHEESECLOTH proved the vulnerability of OpenSSL in approximately 37 hours, using 460 GB of protocol communication.

6.4 Optimizations

Tab. 2 contains the improvements yielded by our key optimizations (Sec. 4). We ran the GRIT and FFmpeg case studies with each optimization disabled and report on the number of multiplication gates in the resulting ZK circuit. In addition, we provide the percentage improvement over the baseline numbers from Tab. 1. The public-pc optimization reduced gate size by 37% in the shorter GRIT execution and 6% for FFmpeg. While this is an improvement, these results indicate there is still room for improvement in our analysis that determines the number of public segments to generate for longer executions. The sparsity optimization with $s = 2$ offers modest improvements of 4%–5% in gate size.

7 Related Work

Recent work has provided the first, exciting steps toward proofs of vulnerability in ZK. `BubbleRAM` [29] is an efficient framework for proving vulnerability, leverage novel protocols for converting between computations in arithmetic and Boolean fields, efficiently handling both read-only and read-write memory, and the `Stack` protocol [30] for proving satisfaction of circuits with explicit disjunctions. Although our current statement compiler partially overlaps with `BubbleRAM` because it implements an older scheme for modeling RAM computations [10], most of our paper's key contributions, namely simplifying unrolled computations using partial evaluation and the novel scheme for generating statements of application leakage, are largely independent of the contributions of [29, 30], and we believe that the approaches could be composed. In particular, `Stack` was evaluated on code snippets representative of a practical CVE of up to 50 LoC; due to its efficient support of disjunctions, it could scale to prove that one of many more such snippets is vulnerable, but it likely strongly benefit from CHEESECLOTH's program optimizations if any particular code segment increased in size.

Reverie [27] is a framework for proving exploits in microprocessor code, consisting of a circuit generator that compiles a given program to an arithmetic circuit and an instantiation [36] of the "MPC in the Head" protocol [33]. The compiler generates statements from exploits that have been formalized as executing a designated instruction that signals an error condition (i.e., violations of reachability properties, formalized directly in the program's control flow); the evaluation of Reverie demonstrates that it can be used to prove *Capture the Flag* (CTF) exploits that require up to 51K cycles on an MSP430 microprocessor. The core contributions of Reverie are largely complementary to those of CHEESECLOTH, which could potentially be adapted to efficiently compile vulnerability statements about programs in intermediate languages to control reachability properties.

Recent work on static program analysis in ZK [21] has presented techniques for proving over-approximations of all program executions without revealing further details of the program, and instantiates the framework on an abstract domain for information flow based on taint tracking. The static analysis itself is designed to prove that a program may leak information: thus, it cannot yield results that directly imply that a program must leak, although in many cases it could provide evidence that could strongly inform an analysts belieft that a program may in fact leak.

Our MicroRAM machine is inspired by TinyRAM [10] but departs from their design in sevaral important ways discussed in Sec. 3.1. There are also some key differences in scope and capabilities. TinyRAM is designed to express correctness of any nondeterministic computations while MicroRAM focuses on vulnerable programs. For example, *SNARKs for C* [10] approach cannot encode proofs of memory-safety vulnerability in ZK directly. Instead, they encode knowledge of the existence of a complete, concrete vulnerability trace, which includes copies of exact values in all local variables and the values in memory at each point in the trace and the bug must be evident in the execution's return value. Our approach encodes memory vulnerabilities directly, resulting in a significantly more succinct witnesses to vulnerability. In particular we can disregard the trace after the bug is found and we don't rely on the programs return value.

Furthermore, TinyRAM approach does not scale to proofs of vulnerabilities in practical programs and has only been evaluated on programs with less than 1,200 low-level instructions [10]. In contrast, the optimizations proposed in this work enables us to support programs with more than 340,000 lines of low-level code (Tab. 1). Beyond scalability, MicroRAM supports a much broader subset of the C language, including most of the standard C library.

Pantry and Buffet [17, 52] represent computation as arithmetic constraints; a solution to the constraints is a valid trace of the computation. After implementing the memory consistency approach of TinyRAM, they report results orders of magnitude better than TinyRAM. Buffet supports all features in the C language, with the exceptions of `goto` statements and function pointers. To translate computation into a constraint system, Pantry and Buffet must unroll loops to publicly revealed bound (although the original work does not explicitly discuss encoding recursive functions, we hypothesize that they would be encoded similarly, using bounded function inlining). The constraint system must include every branch of conditionals and every iteration of every loop (multiplicatively with nested loops) which could lead to blowups in the constraint system, however the authors suggest that this would only happen in degenerated cases and would not be common in practice. A variant Pantry/Buffet that uses zero-knowledge techniques to keep the state private with the same efficiency benefits. When presenting our approach, we compare facts about private inputs that it reveals to those revealed from public loop bounds (Sec. 3.2.3).

vRAM [53] has achieved further efficiency with a ingenious universal preprocessing that allows the parties to use a smaller circuit tailored to verifying the specific program on the chosen inputs. Unfortunately, such tailored circuits can reveal significant information about the input provided. Our public-pc optimization (Sec. 4.1) attempts to balance the gains of a tailored circuit and the privacy requirements of the prover.

8 Conclusion

Due to a sustainted successes in the development of ZK protocols, recent techniques have reached the cusp of proving knowledge of realistic vulnerabilities and proving subtle exploits in low-level code. This paper describes how a host of core techniques from compiler design—namely, conservative instruction profiling and under-approximating information-flow tainting—can be implemented in an optimizing proof-statement generator to produce proofs of vulnerability in commodity software that can be triggered only be using a considerable amount of time and space.

Our practical experience has produced a zero-knowledge proof of memory unsafety in FFmpeg and a proof of leakakge in OpenSSL that directly used the Heartbleed exploit as a witness and demonstrates that zero knowledge proofs of vulnerability in critical application software are now practical.

Availability and Ethical Considerations

We are in process of open sourcing the implementation of CHEESECLOTH for publication and artifact evaluation. CHEESECLOTH aids in responsible disclosure by producing zero-knowledge proofs of the existence of vulnerabilities while keeping the vulnerabilities and exploits secret. All vulnerabilities used in our evaluation have been previously disclosed publicly, and fixes are widely deployed. Thus, the work presented in this paper does not constitute an unethical disclosure of potentially harmful information.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0085. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA). Approved for Public Release, Distribution Unlimited.

References

[1] CVE-2013-0864. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0864. Accessed: 2022-10-10.

[2] FFmpeg. https://ffmpeg.org/. Accessed: 2022-09-01.

[3] OpenSSL: Cryptography and SSL/TLS toolkit. https://openssl.org/. Accessed: 2022-09-05.

[4] Picolibc: C libraries for smaller embedded systems. https://keithp.com/picolibc/. Accessed: 2022-10-10.

[5] The Heartbleed Bug. https://heartbleed.com/. Accessed: 2022-09-05.

[6] zkInterface: SIEVE intermediate representation (IR) proposal. https://hackmd.io/@danib31/BkP9HBp2L. Accessed: 2022-10-10.

[7] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[8] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[9] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Malkin and Peikert [37], pages 92–122.

[10] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[11] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. TinyRAM architecture specification, v0.991. https://www.scipr-lab.org/doc/TinyRAM-spec-0.991.pdf, 2013.

[12] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, 2014.

[13] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Notices*, 39(1):14–25, 2004.

[14] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 168–197, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany.

[15] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Time- and space-efficient arguments from groups of unknown order. In Malkin and Peikert [37], pages 123–152.

[16] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part I*, volume 11272 of *LNCS*, pages 595–626, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Heidelberg, Germany.

[17] Benjamin Braun, Ariel J Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 341–357, 2013.

[18] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[19] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[20] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[21] Zhiyong Fang, David Darais, Joseph P Near, and Yupeng Zhang. Zero knowledge static program analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2951–2967, 2021.

[22] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 178–191, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[23] Galois, Inc. swanky: A suite of rust libraries for secure computation. https://github.com/GaloisInc/swanky, 2019.

[24] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[25] Joseph A Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE, 1982.

[26] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38(3):690–728, 1991.

[27] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *Cryptology ePrint Archive*, 2022.

[28] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[29] David Heath and Vladimir Kolesnikov. A 2.1 khz zero-knowledge processor with bubbleram. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2055–2074, 2020.

[30] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 569–598. Springer, 2020.

[31] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. In *2021 IEEE Symposium on Security and Privacy*, pages 1538–1556, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[32] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 150–169, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

[33] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30, 2007.

[34] Jasper Vijn. GRIT: Gba raster image transmogrifier. https://github.com/devkitPro/grit, 2022.

[35] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[36] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 525–537, 2018.

[37] Tal Malkin and Chris Peikert, editors. *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

[38] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. Sublinear zero-knowledge arguments for RAM programs. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 501–531, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[39] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[40] Andrew C Myers and Barbara Liskov. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review*, 31(5):129–142, 1997.

[41] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

[42] James Parker, Niki Vazou, and Michael Hicks. Lweb: Information flow security for multi-tier web applications. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[43] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[44] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[45] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[46] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[47] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1997.

[48] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[49] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, pages 95–106, 2011.

[50] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 39(11):85–96, 2004.

[51] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*, San Diego, CA, USA, February 8–11, 2015. The Internet Society.

[52] Riad S Wahby, Srinath TV Setty, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[53] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 908–925. IEEE, 2018.