SNARKProbe: An Automated Security Analysis Framework for zkSNARK Implementations

Yongming Fan[1], Yuquan Xu[1,2], and Christina Garman[1(✉)]

[1] Purdue University, West Lafayette, USA
{fan322,xu1210,clg}@purdue.edu
[2] Georgia Tech, Atlanta, USA

Abstract. With the growing interest in privacy-enhancing technologies, we are seeing a complementary growth in the desire to build and deploy complex cryptographic systems that involve techniques like zero-knowledge proofs. Of these, general purpose proof systems like zkSNARKs have seen the most interest, due to their small proof size, fast verification, and expressiveness. Unfortunately, as we have seen with many areas of cryptography, guaranteeing correct implementations can be tricky, as the protocols themselves are complicated and often require substantial low-level manual effort to achieve maximum performance. To help with this problem, and gain better assurances about the correctness and security of already implemented zkSNARK protocols and the privacy-enhancing applications that use them, we design and build `SNARKProbe`, an automated security analysis framework for zkSNARKs that can scan R1CS-based libraries and applications to detect various issues, such as edge case crashing, cryptographic operation errors, and/or inconsistencies with protocol descriptions. `SNARKProbe` leverages a variety of analysis techniques, including fuzzing and SMT solvers. We test the performance of `SNARKProbe` on a variety of different experimental parameters to demonstrate its practicality and reasonable runtime, and we also evaluate its ability to find potential inconsistencies and errors in implementations.

Keywords: Cryptography · zkSNARKs · automation · software security

1 Introduction

We have seen a growing interest in privacy and privacy-enhancing technologies from the general public [49], which has led to subsequent increased interest from parties that build and deploy the technologies that we use every day, with even the US government expressing interest in ways to best deploy privacy-enhancing technologies for data analytics [45]. One of the key components in many privacy-enhancing protocols are zero-knowledge proofs [37], which are cryptographic algorithms that allow one party (the prover) to prove to another (the verifier) that a statement is true, without revealing any information beyond the

C. Pöpper and L. Batina (Eds.): ACNS 2024, LNCS 14584, pp. 340–372, 2024.
https://doi.org/10.1007/978-3-031-54773-7_14

validity of the statement itself. Of these, one of the most popular instantiations are zkSNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) [26,39,52], because of their small proof size, fast verification, and expressiveness.

Because of this popularity, we have seen an explosion of new zkSNARK protocols and libraries being developed in academia, and substantial interest from industry and other domains in actually deploying these protocols for real world usage [6–8,19,20]. Unfortunately, they can be quite difficult to implement correctly, as the protocols themselves can be complicated and involved, and much of the work to generate a single proof for a single application is often done manually and hand-tuned to ensure maximum performance, with developers often working at the circuit or gate level to design the best protocols. Additionally, as we have seen in the past, deploying complex cryptography has not come without its challenges [35,48,51,56], and applications that use zkSNARKs have not been immune to these either, as Zcash for example has found various cryptographic bugs in different components [1,2,4]. And checking any cryptographic implementation manually can be time consuming and potentially error prone.

While automated techniques like fuzzing have been used to test cryptographic implementations specifically before [21], we cannot use existing cryptographic fuzzers here as they generally test only for known weaknesses in certain schemes and are unable to produce the types of inputs that we need in a cost-effective way. Additionally, most general purpose fuzzing tools cannot detect what we refer to as *cryptographic logic errors*, i.e., errors that might result in an incorrect computation but not a program crash, particularly with regards to zkSNARK libraries.

All of this motivates us to ask the question:

Can we develop better tooling to automatically check the security of and precisely locate software bugs and cryptographic logic errors in the proof generation processes and libraries of zkSNARK protocols and the applications that use them?

To help answer this question in the affirmative, we design and build `SNARKProbe` to automatically both check the correctness and consistency of proof programs generated by R1CS-based zkSNARK libraries, as well as inspect the security of the libraries themselves and flag any inconsistencies between a protocol's implementation and its description, no matter how minor[1]. Our primary goal is to make the process as automated as possible, thus reducing the chance of human error in the process, as well as lowering the barrier to entry for usage. We wish to enable even those who might not be cryptographic experts to inspect a library or proof implementation without understanding details of the underlying protocol or specifics of the library. Users need only indicate some configuration settings such as the fuzzer parameters and expected proof statement, and our tool will automatically analyze the library and proof program to detect possible implementation and cryptography related errors.

[1] Note that this is something that professional security audits do flag, as these can lead to potential future problems, see [4].

In order to achieve this, we utilize a combination of techniques. Dynamic analysis allows us to trace real-time data and variable values, as well as handle a variety of different zkSNARK libraries written in different programming languages without needing additional (manual) language specific adaptations. Custom fuzzing techniques allow us to produce a variety of valid or invalid R1CS matrices (i.e., proofs) to exercise the different codepaths in a library. SMT (Satisfiability Modulo Theory) solvers allow us to help verify the consistency of a user-specified set of statement equations (i.e., the desired proof) with the actual R1CS matrix used in the application. And the notion of ideal files and a value checker helps ensure that a library correctly realizes the given underlying protocol.

While one could tackle some of this from a formal analysis approach [15], we view `SNARKProbe` to be complementary, as formal analysis works best for specific, static protocol implementations (like libraries), and we wish to also be able to check application-specific proofs. Formal verification in such a scenario would need to be done for *each individual proof*, which is both time-consuming and requires expertise in such techniques for every project that wishes to use a zkSNARK. `SNARKProbe`, on the other hand, is designed to work with a variety of different protocols and to be easy to run on a given proof implementation in an application, without requiring substantial expertise. We will see later on how a formally verified library for a specific protocol could be adapted as part of `SNARKProbe` to provide stronger guarantees, though we leave this for future work. As such, we believe `SNARKProbe` is a step in the right direction for ensuring the correctness of existing zkSNARK implementations and applications.

We choose to focus on two widely used zkSNARK protocols, Pinocchio [27,52] and Groth16 [39], and four libraries, `libsnark` [3], `Bellman` [10], `arkworks` [14], and `gnark` [13]. `libsnark` implements both Pinocchio and Groth16 in C++, while `Bellman` and `arkworks` implement Groth16 in Rust, and `gnark` implements Groth16 in Go. These represent four of the most popular and widespread open source libraries that have seen real world usage and underpin many projects that use zkSNARKs. This selection also demonstrates the flexibility of `SNARKProbe` to handle different R1CS-based zkSNARK protocols, as well as diverse languages.

In addition to designing and implementing `SNARKProbe`, we tested its performance extensively, using a number of different statement types to test the Constraint Checker, and a wide variety of R1CS matrix sizes and configurations (which are representative of statements of varying complexity) to test Snark-Fuzzer. We demonstrate performance that we believe is both reasonable and scalable for existing zkSNARK use cases[2]. We also evaluated `SNARKProbe`'s ability to detect potential inconsistencies or different types of errors. While we did not find any new exploitable errors, we did find a number of issues across the various libraries that we consider to be potentially unsafe or that might require special care by a developer to navigate correctly.

Our Contributions. In this paper we make the following contributions:

[2] And, in fact, in one instance the underlying library was actually unable to scale and run before we ran into any issues with our tool.

- We design SNARKProbe, a tool that can automatically check for potential software errors and cryptographic logic errors, as well as for consistency in the implemented proof statement, in R1CS-based zkSNARK libraries.
- We implement and build SNARKProbe[3] for two zkSNARK protocols, Pinocchio [27,52] and Groth16 [39], and four libraries, libsnark [3], Bellman [10], arkworks [14], and gnark [13], to demonstrate its flexibility and extensibility.
- To the best of our knowledge, we are the first to explore applying fuzzing and other analysis techniques to zkSNARKs specifically.
- We evaluate the performance of SNARKProbe in an extensive set of experiments, demonstrating its acceptable performance for real world applications.
- We demonstrate and discuss SNARKProbe's ability to *automatically* catch different types of errors in zkSNARK libraries, finding seven inconsistencies and successfully locating a prior CVE.

Possible Ethical Considerations. While our tool did find a few inconsistencies in gadgets and protocol specification versus implementation throughout our testing, all of these issues were either not exploitable, already known, or just require care from a developer when implementing a proof, and as such we do not believe there is anything to disclose.

2 Background and Related Work

In this section, we introduce relevant background and prior work.

zkSNARKs. In a zero-knowledge protocol [37] a user (the prover) proves a statement to another party (the verifier) without revealing anything about the statement other than that it is true. zkSNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) [26,39,52] are one of the current most popular zero-knowledge proof systems, and have begun to see increasingly complex, real world deployment in a number of privacy-preserving applications [6–8,19,20]. At a very high level, zkSNARKs work as follows. The first step in proof creation is to turn the statement that one wishes to prove into an equivalent form that relies on knowing a solution to some algebraic equations. This representation is then broken down into an arithmetic circuit. To ensure the correct evaluation of the circuit (and thus the proof), a programmer must express a series of constraints on the wires, called a constraint system, which is typically a Rank 1 constraint system or R1CS. zkSNARK libraries often provide an abstraction called a *gadget*, as expressing large programs with constraints can be quite difficult. A gadget allows programmers to specify a series of inputs, hidden internal variables and constraints on the inputs and internal variables. Rather than expressing a large program as many constraints, one can instead express it as some gadgets, and a few constraints binding them together.

Fuzzing. Fuzzing is an automated software security testing technique to explore software for bugs that cause incorrect or unexpected results, program crashes,

[3] https://github.com/BARC-Purdue/SNARKProbe.

Table 1. Comparison of `SNARKProbe` and a selection of cryptographic and generic fuzzing tools. A ◐ means achieves in select instances.

Fuzzer	Adapt to zkSNARK	Error Types			Error Location	
		Program	Logic		Code	Scheme
			Crypto	zkSNARK		
AFL [60]	●	●	○	○	●	○
CDF [21]	○	●	◐	○	○	○
SHA-3 Test [46]	○	●	●	○	○	○
TLS-Attacker [57]	○	●	◐	○	●	◐
`SNARKProbe`	●	●	●	●	●	●

or in some extreme cases, lead to exploit paths. Fuzzing works by generating inputs and monitoring the program for crashing, assertions, and memory leaks [30] without the need for manual review by developers, security engineers, and auditors. An important part of fuzzing is code coverage, a metric used to evaluate how much of the target program is tested. Early fuzzing systems only generated inputs by mutating a seed input and watching for crashes or errors. Over time, fuzzing test structure has been improved and other techniques have been developed to improve the effectiveness [28,29,36,38,40,59,60].

SMT Solvers. Satisfiability modulo theory (SMT) is a complex version of the boolean satisfiability problem (SAT) that can determine if a mathematical equation is satisfiable or unsatisfiable. Z3 (which we choose to use in this paper) is an efficient SMT solver that has been used in various software verification and analysis applications [47]. Typically, SMT solvers such as Z3 can only handle first-order logic, but recent research [23] proposed a pragmatic extension for SMT solvers to support higher-order logic.

2.1 Prior Work

There has been work to automate the conversion between proof statement and R1CS [33,41,42,44], but they do not necessarily guarantee the correctness of the translation. Manually writing/editing R1CS is also still popular as it can result in substantial efficiency gains. Such work can help secure new applications but cannot help with the security of existing implementations.

Fuzzing has been used for some cryptographic implementations. Special-purpose fuzzing has been used for TLS [57,58]. CDF [21] targets cryptographic algorithms such as DSA, ECDSA, and RSA. We are inspired in part by these ideas, though we have no known test vectors or reference implementations to compare against and use more targetted input generation techniques.

We provide a comparison of `SNARKProbe` and the most relevant existing fuzzing tools in Table 1, focusing on the types of errors they are able to catch, if they can precisely locate such errors, and their adaptability to zkSNARKs. Currently, most fuzzing tools are limited to identifying software errors. While some tools such as CDF and SHA-3 Test have the capability to identify cryptographic logic errors, their functionality is restricted and cannot be easily extended. Presently, our SnarkFuzzer stands as the only fuzzing tool capable of detecting both software and cryptographic logic errors for zkSNARKs, accompanied by source code references and protocol location information.

Additionally, an important part of many fuzzers is how they handle code coverage analysis. While off-the-shelf tools like LLVM might seem to fit well for this, as it natively provides a source code line coverage data report [11], this is insufficient for our desired application. Our fuzzer focuses specifically on branch coverage and coverage of critical cryptographic components, not generic line coverage. Additionally, we found that many existing zkSNARK libraries may not work well (or even compile) with LLVM. This motivates our development of the Branch Model and its use of GDB, as existing tools do not suffice.

There has also been prior work on automating the development of cryptographic attacks. Much of this work has focused on side-channels [22,53,54], though it has also seen other applications [25]. Prior work has also sought to automatically deploy existing, known attacks [43]. These are largely orthogonal and do not extend to existing zkSNARK systems.

There is a growing trend towards computer aided cryptography and formal verification of implementations (see [24] for a detailed discussion). Project Everest [15] contains a variety of formally verified software components. While we are seeing increasingly complex protocols being formally verified [55], we have not yet seen these techniques applied to zkSNARKs. We see our work both as a step towards a formally verified zkSNARK implementation, bridging the current gap through the use of heuristic techniques like fuzzing, as well as a complement to it. A formally verified implementation could be used to provide a trustworthy ground truth for our fuzzing techniques, for example. In addition, formal verification is typically targeted at a single implementation, which necessitates re-running the often complicated process for each new application. SNARKProbe is designed to streamline the process and make it easy for a user to check a variety of applications or libraries, with little manual effort.

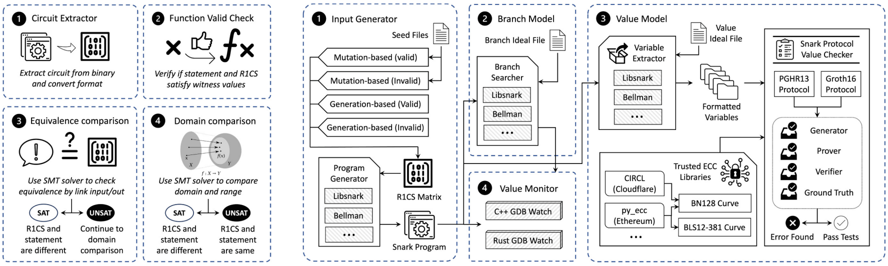
3 Overview

Before diving into the details of our design, we give a brief high-level overview of our SNARKProbe system, its workflow, and the intended use cases and users.

3.1 Users and Use Cases

We view the targeted users of SNARKProbe to fall into three main categories: 1. An application user that wishes to evaluate if a given (open-source) application correctly implemented their specified proof statement without having to dig through source code or understand the protocol; 2. Developers who want to build applications using zkSNARKs and wish to gain assurances in the correctness of a library and that a proof they built correctly realizes a given statement;

and 3. Security engineers that want to automatically scan a library/application and detect various issues, such as edge case crashing, cryptographic operation errors, and/or inconsistencies with protocol descriptions, allowing them to focus their time and manual effort on potential issues. SNARKProbe helps achieve all these use cases by outputting a full evaluation report that includes a notion of security confidence, any discovered (known) security vulnerabilities or warnings, and potential risks of the target library or proof.



(a) Constraint Checker

(b) SnarkFuzzer

Fig. 1. Workflow of SNARKProbe. White boxes represent subcomponents universal for all zkSNARK libraries, forward diagonal stripes represent those designed for a specific library, and backward diagonal stripes represent those designed for a specific programming language.

3.2 Workflow

We now outline the workflow of running SNARKProbe, as well as establish terminology, before presenting the details and implementation in subsequent sections. SNARKProbe is composed of two broad sub-tools: the Constraint Checker and SnarkFuzzer. These components can operate both independently of each other as well as in sequence, depending on what the user wishes to check.

Recall that at a high level, a zkSNARK proof is generated in two main phases. First (*circuit generation*), the user has a *(proof) statement* that they wish to prove about, which requires converting this statement into a *circuit* that the zkSNARK protocol understands. *R1CS* is a specific form of this that we focus on in this work. Second (*proof generation*), a zkSNARK library takes the circuit (R1CS) as input and produces a proving key, verification key, and a *proof.*

The first step verifies the consistency of a user-specified set of proof statement equations with the actual R1CS matrix used in the application, to ensure that the circuit generation process generates the same proof as the developer states. The Constraint Checker uses an SMT solver to compare the input values, output values, domain, and range of the given statement equations with R1CS matrix to find any errors in the conversion. Figure 1a shows the internal steps of the Constraint Checker, which we discuss in detail in Sect. 4.1.

The second step checks the correctness of the library and proof generation process. Figure 1b shows the steps of SnarkFuzzer, which we discuss in detail in Sect. 4.2. First, the Input Generator produces an R1CS matrix (i.e., proof) using a variety of fuzzing techniques and compiles this into an executable proof program with the selected library, which serves as the input for the Branch Model. Then, the Branch Model investigates the visitation status for each branch during execution and provides feedback to the Input Generator to determine if another input is needed. This coverage feedback fuzzing helps improve the coverage rate of SnarkFuzzer, which improves the chance of catching errors in the target library. Finally, the Value Model re-evaluates the protocol calculations to detect any potential cryptographic logic errors and looks for inconsistencies caused by unexpected value changes or implementation errors.

Together, these two components allow SNARKProbe to evaluate the end-to-end correctness and security of a specific implementation of a given proof statement for a given library (or to individually check different parts if desired).

Unlike standard fuzzers, SnarkFuzzer can detect both software and cryptographic logic errors. We consider a *software error* to be a bug or fault in the software that causes unintended behaviors or incorrect performance, such as crashing, overflow, or a system error. A *cryptographic logic error*, on the other hand, is one that causes a program to produce an incorrect result due to errors in the cryptographic protocol implementation, such as incorrect mathematical operations or disregarding the specification. A cryptographic logic error will not cause a software crash or raise an error message, making them much harder to detect. For a basic example, in RSA, software errors might cause a program crash, but cryptographic logic errors are issues where the ciphertext value was not computed correctly ($m^e \mod N$ computed incorrectly) or too large a ciphertext was used (e.g., larger than the modulus).

While we currently focus on R1CS-based zkSNARKS, we believe this high level design can be adapted without major changes to support additional (newer) types of proofs, particularly in the case of SnarkFuzzer, which would largely only involve changing the input generation methodology and format.

Why Dynamic Analysis? Dynamic analysis supports our goal of finding cryptographic logic errors by allowing us to trace and extract real-time data and variable values, which we can then use to check for cryptographic protocol conformance and implementation errors. It also allows us to easily support a number of different zkSNARK libraries written in different languages without the need for additional manual intervention, extensions, or program instrumentation by the user, making SNARKProbe flexible as well.

4 Design and Implementation

In this section, we introduce both the high level goals and concrete implementation decisions, tools, and techniques that allow us to realize SNARKProbe. We start with the Constraint Checker and then discuss the various components of SnarkFuzzer. These two sub-tools can be used either in conjunction with each other, or separately, depending on what the developer wishes to check.

4.1 Constraint Checker

Goals: *The Constraint Checker helps ensure that the realized proof is equivalent to a specified statement. This also allows the Constraint Checker to find errors in flattening and/or gadget use. It achieves this by comparing the equivalence of a user-specified proof statement and the compiled R1CS gates to ensure that the proof generated by the library represents the developer's specified statement*[4].

Design. The first step in generating a zkSNARK proof is converting the desired proof statement equations into R1CS. These R1CS gates should have the same domain, range, and output value as input statement equations. To check this, we developed our Constraint Checker, which leverages an SMT solver and associated techniques to verify the equivalence of R1CS gates to the specified proof statement functions[5]. To run the tool, the user only needs to write a short script to define the statement function, primary inputs, and auxiliary inputs in their zkSNARK proof, and everything else then happens automatically. Figure 1a shows the key steps in the Constraint Checker, and we also provide an example in Appendix A.3. We note that our SMT solver could be replaced by any other techniques that allow one to check for function equality and fit the desired goals.

Implementation. The Constraint Checker contains two major components—the Circuit Extractor and a three step equivalence check—which we discuss now. We discuss optimizations in Appendix A.1.

Circuit Extractor. The Circuit Extractor can extract R1CS gates with private inputs, witness, and gadget relations from an executing zkSNARK binary program. We do this using GDB. We set a GDB breakpoint at the circuit declaration line, convert the circuit to an R1CS matrix when the program reaches the breakpoint, and save the formatted R1CS matrix for equivalence comparison. Automatic extraction both makes the tool more usable and reduces the potential for human errors in the equivalence comparison. Many zkSNARK libraries use Montgomery modular multiplication as an optimization. We also developed a Montgomery representation translator that can convert the Montgomery form to a standard integer format, so that our circuit extractor can produce a z3py (our SMT solver of choice) readable integer format.

After extracting the circuit, the Constraint Checker uses the SMT solver to verify the equivalence of functions through a three step process: a function valid check, equivalence comparison, and domain comparison. If one or more tests fail,

[4] We note that we cannot do anything about the garbage-in-garbage-out problem (for example, the case where a developer implements a proof that misses a necessary case, such as omitting checking if a value is greater than 0). We can just help determine if the specified statement and realized proof are likely equivalent.

[5] We work around the general undecidability of function equality by operating in the more constrained setting (over the finite fields) and adding special rules for primary variables and auxiliary variables.

then we determine that the given statement equation is not equivalent to the produced program. Otherwise, then we can provide assurances that it is highly likely that the proof realizes the specified statement correctly[6].

Function Valid Check. As the witness should be a solution in both the statement equations and equations built by the R1CS gates, we first create two SMT solvers S_1 and S_2 to verify that fact (see Appendix A.2 (b)). The Constraint Checker considers the statement equations and R1CS gates to not be equivalent if the witness is not a valid solution for both.

Equivalence Comparison. The equivalence comparison verifies if the statement equations $y_1 = f(x_1)$ and R1CS gates $y_2 = f(x_2)$ have the same output for all the same inputs. First, the Constraint Checker creates a solver S, and adds $y_1 = f(x_1)$, $y_2 = f(x_2)$, $x1 = x2$ and $y_1 \neq y_2$ to the solver S (see Appendix A.2 (c)). If S results in `SAT`, then the SMT solver has discovered that the statement equations and R1CS gates have a different output for at least one input and thus considers them not equivalent, and otherwise, it results in `UNSAT`. Like in many other use cases, the Constraint Checker cannot guarantee equivalency if the SMT solver returns `UNSAT` or `UNKNOWN`. Note that the SMT solver may return `UNKNOWN` for a variety of reasons, including running out of memory, or if the quantifier-free fragment is undecidable (such as nonlinear integer arithmetic) or too "expensive" (such as finding primes $p * q = N$ for some N).

Domain Comparison. The SMT solver cannot directly calculate an equation's domain, but it can find the maximum and minimum values for an equation (Fig. 4a in Appendix A.4 shows an example of different upper and/or lower bounds). We also use the SMT solver to find out if there is a gap between the domains' of the statement equations and R1CS gates, though we cannot guarantee to find all such gaps that may exist (see Appendix A.4 for a detailed discussion).

4.2 SnarkFuzzer

Goals: *SnarkFuzzer is a smart fuzzing tool with the primary goal of finding potential logic or software errors in zkSNARK libraries. It produces zkSNARK programs with different proofs as input, to detect and catch these errors and provide full (important) code coverage.*

SnarkFuzzer is composed of four major subcomponents: the Input Generator, Branch Model, Value Model, and Value Monitor. The Input Generator is responsible for generating an R1CS matrix and converting it to a zkSNARK program as input for the target library. The Value Model and Value Monitor are designed to detect implementation and cryptography related errors. The Branch Model helps manage branch coverage and provides feedback to help SnarkFuzzer decide

[6] We cannot *guarantee* equivalence due to the fact that the SMT solver might output `UNKNOWN`, as well as possible domain gaps (see Appendix A.4).

if it needs to generate another input program. Figure 1b shows the structure of SnarkFuzzer with both its high level design as well as its implementation details.

SnarkFuzzer: Input Generator

Goals: *The Input Generator produces valid or invalid R1CS matrices and compiles them into executable proof programs for use by other components as testing inputs. This should be done in a "smart" manner to help better catch errors.*

Design. The first step for our fuzzer is the generation of inputs to test a target library. In SnarkFuzzer, these inputs are simply valid or invalid R1CS matrices that represent a variety of (possibly random) proof statements. To improve the chances of catching potential logic or software errors in the library, we desire this input generator to be "smart", in the sense that it should not just randomly generate inputs, but should take into consideration both the concept of valid/invalid proofs, as well as proof or R1CS edge cases. At a high level, the Input Generator generates an R1CS matrix, and then the Program Generator uses this matrix to produce and compile an executable zkSNARK program with a given library. The compiled program will then be used as input for other components, like the Branch Model, Value Model, and Value Monitor. Before generating the R1CS matrix, the Input Generator needs to decide on the size of the generated matrix (number of rows n_r and columns n_c) and the number of public inputs n_p. This can be done by simply choosing randomly, or in a smarter way by choosing from a specified distribution. The Input Generator can be instantiated with any type of fuzzer that meets the aforementioned goals, giving it the flexibility and adaptability to use new techniques and tools, as they are developed.

Implementation. We developed four different methods of input generation to produce these matrices based on mutation-based and generation-based fuzzing. These methods are complementary and all work in tandem to generate the fuzzing inputs (the user can specify the percentage of inputs generated by each method). Each method targets a specific type of R1CS matrix to provide better overall coverage and comes with its own set of unique benefits (and trade-offs).

Generation-Based Invalid Matrix. Our generation-based fuzzer is designed to create an R1CS matrix from scratch. Generation-based fuzzing has the advantage of not depending on input seeds, allowing it to function without (user) input. The random number generator will generate $n_r * n_c$ integers as an R1CS matrix and $n_r - 1$ integers as the witness w (since the first value of the witness is always equal to 1), and we use the constraint relation $w \cdot A \times w \cdot B - w \cdot C = 0$ to verify the ground truth of the matrix. This tests a library with possible but unrealistic examples to search for edge case errors that are hard to reach with traditional examples.

Generation-Based Valid Matrix. To produce a valid matrix that simulates a real world proof example, the random number generator will generate $n_r - 1$ integers as a witness w. Then, a valid matrix will be calculated with z3py based on this witness and the constraints relations in $w \cdot A \times w \cdot B - w \cdot C = 0$. Depending on the actual values in the randomly chosen witness, the complexity of solving the equations to fill out the matrix may vary, but a larger matrix (i.e., larger witness) usually results in a longer generation time. As such, we recommend that for testing large R1CS matrices, the generation-based fuzzer is run once to generate an input seed that can be given to the mutation-based fuzzer.

Mutation-Based Invalid Matrix. The mutation-based fuzzer requires an R1CS matrix with a number of public variables as a seed, and then flips values to generate a new matrix. We use Atheris, a mutation-based, coverage-guided fuzzer developed by Google [9]. The seed is converted into a byte array, then Atheris will flip the array, and then it is converted back to a (new) R1CS matrix. Because we have randomly flipped values in the matrix, the R1CS constraint relations will be destroyed, and the matrix will be invalid. This technique can work off of and mutate input examples that might be hard to produce by generation-based fuzzing.

Mutation-Based Valid Matrix. Our mutation-based fuzzer can also produce a valid R1CS matrix. Instead of flipping random values, a random number c will be generated, and the new matrix M will be calculated based on the seed matrix M_0 as $M = cM_0$. This multiplication will not break the constraint relations, so the new matrix is still valid. This can be used with the generation-based fuzzer to produce valid large R1CS matrices in a short time, thus increasing performance.

SnarkFuzzer: Ideal Files

Goals: *The ideal files act as a guide for the Branch Model and Value Model to check code coverage and protocol calculations. They contain information related to the source zkSNARK library such as branch locations, variable names, and data types.*

Design. Each library and protocol needs two ideal files: an ideal branch file that contains the locations of the branches in a library (automatically generated), and an ideal value file that contains information about the variables in a program (manually generated). Manually creating the ideal value file does not require any specialized knowledge related to zkSNARK protocols, the library, or cryptographic techniques, and the user only needs basic skills such as defining functions, conditions, and assigning variables. These ideal files can be reused for any program using the same protocol and library.

Ideal Branch File. An ideal branch file contains protocol-specific information related to the branches in a library. Function calls, if-conditions, and loops are all considered branches. The branch file also contains information about IMPORTANT versus UNIMPORTANT branches to help the Value Model. Important branches include cryptographic calculations and other core protocol components that we wish to ensure testing and coverage of, while unimportant branches are those that do not involve relevant computation.

Ideal Value File. An ideal value file contains location information for all variables that must be used to evaluate the protocol calculation. The Value Model will use line and path information in this file to extract all variables' values. Only variables defined in the official protocol paper should be added to the ideal value file.

Implementation. We have already developed the ideal files for Pinocchio and Groth16 in `libsnark` and Groth16 in `Bellman`, `arkworks`, and `gnark`.

Ideal Branch File. By default, all branches are marked as IMPORTANT and thus covered. We asked four researchers in our group (including people with experience using zkSNARKs and those with only programming experience) to label branches as IMPORTANT or UNIMPORTANT. To ensure SnarkFuzzer does not miss any critical components, only branches labeled as UNIMPORTANT by all four researchers could be marked as UNIMPORTANT in our example ideal branch files[7]. We provide an example snippet of an ideal model file for `libsnark` in Appendix B.1.

In order to find uncallable functions, we used a static analysis tool called Doxygen [12] to draw the library functions' call graph. Doxygen can help a user generate the call graph for a library, and by using this call graph, the user can easily find unused and uncallable functions to reduce fuzzing costs.

Ideal Value File. The ideal value file contains "official" variable names, variable types, and variable names, path, and line number (the final assignment location for a variable) in the library's source code. We provide an example snippet of an ideal value file for the `libsnark` library in Appendix B.1. While line numbers might change as code is maintained, one should not need to manually create a new file often. If there are only minor or non-cryptographic related changes, our tool will automatically match the new line numbers.

SnarkFuzzer: Branch Model

Goals: *The Branch Model monitors and records branch visitation status as SnarkFuzzer runs, and informs and stops it after all IMPORTANT branches have been covered.*

[7] Incorrectly labeling UNIMPORTANT branches as IMPORTANT only increases the fuzzing time without affecting the accuracy of the analysis.

Design. The Branch Model is used to evaluate if the target library has been fully tested. Traditional fuzzers often work by examining utilized memory bytes in a binary or lines of source code hit to check for program coverage, and then mapping this coverage to an input to determine the next one. However, in a zkSNARK library, it is difficult to correlate the exact relation between a specific branch and values in the input program that could trigger it. Therefore, we instead use the idea of function and condition branches to evaluate coverage.

Since any unvisited branches may cause errors, the Branch Model evaluates each input program to acquire the list of branches that the program visited. The Input Generator will continuously generate new program inputs with different proof statements/matrices until all `IMPORTANT` branches have been visited.

To ensure that we achieve both breadth and depth in code coverage, the Branch Model also works to ensure that each branch is covered multiple times with different inputs, keeping a count of the number of times each `IMPORTANT` branch is hit. We allow the user to define a visitation threshold to balance performance with coverage and confidence. As more inputs are generated, we have a higher chance of finding potential issues in the target library or, conversely, more confidence that the target library is safe, at the cost of runtime.

Implementation. Our Branch Model uses GDB breakpoints to investigate the visitation status for each branch defined in the ideal branch file. A GDB breakpoint stops the program whenever the selected location in the program is reached for debugging, and it can be set by line number, function name, or address in the program. When the Input Generator produces a zkSNARK program, the Branch Model sets a GDB breakpoint at the start line for each branch. If a branch is then visited during runtime, GDB will temporarily stop the input program at the corresponding breakpoint. After the program is finished running, the Branch Model records a list of visited branches for the current program. If all important branches have not been visited at least once, or the visitation threshold has not been met, the Branch Model will instruct the Input Generator to generate a new program with a different proof until all important branches have been covered.

SnarkFuzzer: Value Model

Goals: *The Value Model's goal is to find cryptographic logic errors, such as inconsistencies with the protocol definition or incorrect cryptographic operations.*

Design. The Value Model is one of the most important components in SnarkFuzzer, and one of the biggest improvements compared with other fuzzers. Its job is to detect cryptographic logic errors in a library. At a high level, the Value Model must extract all necessary variable values from the input program, which it then feeds to the other components. It achieves this through two major subcomponents: the Value Extractor and Value Checker.

Value Extractor. The Value Extractor extracts variables' values from the source zkSNARK program, which are then used in the Value Checker. The ideal value file provides a list of variables that need to be extracted. Then, the Value Extractor recursively extracts these values from the last assignment location in the program and library, and reformats the values to the Value Checker readable format.

Value Checker. The Value Checker re-evaluates all protocol calculations and compares these ground-truth values to the values extracted from the input program by the Value Extractor to find any possible logic errors. In the re-evaluation process, the Value Checker will faithfully follow the protocol from the formal specification without any optimization, simplification, or reformatting. To build a trustable re-evaluation process, our Value Checker uses ECC libraries that have been tested and widely used. In the future, this component could be replaced by any trusted evaluation source, such as a formally verified implementation.

Implementation

GDB PrettyPrint. GDB can display both the structure and value(s) for a variable. However, for more complicated variables GDB displays some unnecessary variable structure and data. Also, different libraries usually have different data structures (even for the "same" variables), and, as such, the Value Checker cannot directly use the raw, unformatted variable value extracted from the library. For example, Figs. 7a, 7b, and 7c in Appendix B.2 show a G_1 type variable that represents the same number in both `libsnark`, `Bellman`, and `arkworks` but has different structure and value when printed naively.

To make the Value Checker more universal, we developed a PrettyPrint functionality for each library to convert all data structures into the same format. The goals of PrettyPrint are twofold: 1) to allow the extractor to keep only the necessary values for a variable, and 2) to convert the same types of variables from different libraries into a common universal format that the Value Checker can then accept. Figure 7d in Appendix B.2 shows an example of the PrettyPrint result for the same G_1 variable as before.

Value Extractor. The Value Extractor extracts all variables defined in the ideal value file using GDB and PrettyPrint. As the ideal value file contains the line number where each variable is assigned, the Value Extractor can automatically set up a breakpoint for each variable immediately after its assignment. Then, when the input program reaches this, the Value Extractor uses PrettyPrint to save the variable's formatted value.

Value Checker. We developed the Value Checker for Pinocchio by following [52] and [27] and Groth16 by [39] and [50]. To help ensure accurate re-evaluation of the protocol calculations, the Value Checker uses exactly the same variable names as the original papers, and we follow the protocol description exactly, without any optimizations, to avoid introducing mistakes.

ECC Libraries. We use two well-tested elliptic curve libraries to demonstrate diversity and flexibility. py_ecc [31], developed by Ethereum in Python, supports both the BN128 and BLS12-381 curves. CIRCL [32], developed by Cloudflare in Go, supports BLS12-381. Since it is written in Go, we developed the necessary APIs so that our Python-based Value Checker can access the functions. If users would like to test a new zkSNARK library using a curve that py_ecc or CIRCL do not support, they can easily plug it into the Value Checker in this same manner.

SnarkFuzzer: Value Monitor

Goals: *The Value Monitor detects any unexpected value changes after the extraction of initial variable values from the source program.*

Design. While the Value Model extracts all necessary variables' values at the location where the variables are first assigned, and we also need to check if there is a value change after the model extracts these values to ensure consistency between the library and the Value Checker. The Value Monitor can also detect any unexpected side effects caused by the library or dependent functions due to incorrect implementation, out of memory, or other unexpected errors.

Implementation. GDB watch allows SnarkFuzzer to monitor value changes in a debugging session with four CPU debug registers called hardware watchpoints. A hardware watchpoint is very efficient, but it does not support larger data structures given the limited debug registers. Software watchpoints will be automatically applied if GDB tries to setup a watchpoint for a variable that cannot be handled by a hardware watchpoint. Unlike hardware watchpoints, software watchpoints are extremely slow, and watching dozens of variables may require unacceptable processing time. For example, using GDB software watch for a single variable `A_query` in `libsnark` takes more than 30 min.

Therefore, we instead used Valgrind, a dynamic analysis tool that analyzes a program to automatically detect memory management and threading bugs. Valgrind has a gdbserver to simulate traditional GDB hardware watchpoints. Simulation still takes longer than pure GDB hardware watchpoints, but the hardware watchpoint simulation is much faster than GDB software watchpoints. Meanwhile, Valgrind hardware watchpoint simulation does not have any limitation on the number and length of variables, so Valgrind can be used for any zkSNARK source program and libraries with complex data structures.

5 Performance Evaluation

Our first set of evaluations for `SNARKProbe` explores its performance and scalability. All experiments run on an Intel Core i7-8700 CPU @ 3.20 GHz × 12 with 32 GB of RAM running 64-bit Ubuntu 22.04 LTS.

5.1 Performance of the Constraint Checker

We start by testing the Constraint Checker and evaluate its performance on statements with different types of operations and complexities, and present the runtimes in Table 2. There are two runtime results for each experiment. We first recorded the runtime for the SMT solver, which is the most important step in the Constraint Checker, and we also recorded the total runtime, including the processing time of the Circuit Extractor, SMT solver, and other internal evaluations.

Table 2. Constraint Checker runtime for different proof statements

Statement	Equation	Runtime (in seconds)	
		SMT Solver	Overall
Cube	$y = x^3 + x + 5$	0.49	4.18
Cube with Error	$y = x^3 + x + 5$	0.24	3.94
Comparison	$x < 60$	0.54	12.39
SHA-256 Hash	$y = \texttt{sha256}(x)$	0.71	15.39
Inner Product	$\langle u, v \rangle < c_1 \wedge \langle u, v \rangle > c_2$	0.59	18.34

We tested different types of proofs, including the well-known cube example, a comparison (which requires using a gadget), a hash (a complicated mathematics equation), and a proof with inner product, logical operators, and comparison (which requires multiple gadgets and more complex logic). As expected, as we progress in terms of "complexity", runtime generally increases. Interestingly, the hash proof takes the longest in the SMT solver, demonstrating that there are different notions of "complexity" than just statement size.

We also sought to test if the correctness of the statement checked had any impact on performance. We did this by running two different experiments with the cube proof, one where we model a developer correctly flattening the statement equation into R1CS, and a second where we model a mistake in the flattening that introduces an error. Our Constraint Checker took a shorter time to evaluate the incorrect cube program since the Constraint Checker aborts as soon as it finds an error (such as a different domain or output value).

5.2 Performance of SnarkFuzzer

We tested SnarkFuzzer on `libsnark`, `Bellman`, `arkworks`, and `gnark` to evaluate the processing time and performance in different configurations. All experiments run on a single core, except for the Value Monitor which is trivially parallelizable across all cores. We start by evaluating the overall tool to examine the runtime percentage that is dedicated to each sub-component. We then drill down and individually explore the scalability of each sub-component to gain a fuller understanding of SnarkFuzzer's overall scalability.

Overall Runtime with Different Libraries and Protocols. We first explore the proportion of total running time that each different facet of SnarkFuzzer takes, and present the results in Fig. 2.

We ran with five different configurations: `libsnark` with Pinocchio, `libsnark` with Groth16, `Bellman` with Groth16, `arkworks` with Groth16, and `gnark` with Groth16. All of these experiments use generation-based fuzzers to produce valid R1CS matrices (the slowest of the four fuzzing methods we provide) with dimensions in $[30, 60]$. We ran 30 iterations of the Branch Model (since as shown in Fig. 3 that provides reasonable coverage) and calculated the average runtime to complete a full "run"[8] and proof check for each library and protocol.

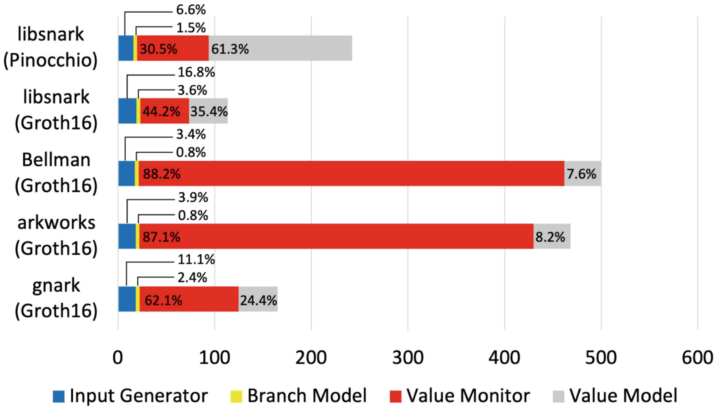


Fig. 2. Runtime of each component in SnarkFuzzer with different libraries and protocols, both in terms of actual time as well as percentage of the total.

We observe that for all libraries and protocols, the majority of time is spent in the Value Monitor and Value Model phases. This is somewhat expected, as the cryptographic operations in the Value Model require time to calculate, and Valgrind's simulation hardware watch also introduces substantial overhead. Since Groth16 has fewer elliptic curve and pairing operations than Pinocchio, testing that involves Groth16 costs less time. However, Valgrind spends a longer time executing a program in Rust than a program in C and Go. Since we use Valgrind in the Value Monitor to find and monitor value changes, we see a longer overall runtime with `Bellman` and `arkworks` than `libsnark` and `gnark`.

We do use small R1CS matrices in these experiments, as the primary goal is to demonstrate the runtime proportion for each component. We make a few key observations about this. We noticed that unless one uses only generation-based fuzzing to produce valid inputs, the size of the R1CS matrix (i.e., the proof) will

[8] We define a "full run" to be using the Input Generator to produce one zkSNARK program, analyzing coverage status with the Branch Model, and looking for any software and cryptographic related bugs with the Value Model and Value Monitor.

not substantially affect the performance of the Input Generator. The Branch Model does not work with the R1CS matrix, so different sizes will result in the same runtime. However, a large matrix will significantly increase the runtime of the Value Model and Value Monitor (see Table 4 for more realistic values). For total runtime reference, an R1CS matrix of size 100 × 5000 takes about 940 s to run the complete SnarkFuzzer with our generation-based invalid matrix fuzzer.

Performance of the Input Generator. Unlike a mutation-based or generation-based fuzzer for an invalid matrix (which are very quick since they only require straightforward operations), generating a valid R1CS matrix with our generation-based fuzzer requires the SMT solver to produce a matrix based on the random witness array. Thus the processing time depends on the matrix size and random value(s) in the witness list, which generally results in a larger matrix taking longer to process. To validate this, we used our various fuzzers to produce matrices of varying sizes (remember that the rows represent the number of constraints and columns the number of variables). In Table 3, we present the processing time for each method (generally under one second for most methods), which confirms our general intuition. We also note that the Input Generator must compile the input matrix (proof) and source code into a binary program, which can require substantial time that is outside of our control.

Table 3. Runtime (in seconds) of the Input Generator to produce R1CS matrices of different sizes.

Matrix Size		5 × 10	20 × 40	50 × 100	50 × 200
Generation Based	Invalid Matrix	<0.01	<0.01	<0.01	0.02
	Valid Matrix	0.16	2.06	23.02	51.27
Mutation Based	Invalid Matrix	0.38	0.49	0.65	0.77
	Valid Matrix	<0.01	<0.01	<0.01	0.02

Performance of the Branch Model. Next, we explore the coverage ability of the Branch Model in terms of breadth, as well as how quickly we achieve a high percentage of branch coverage. We ran 50 iterations of the Branch Model with `libsnark`, which covers more than 85% of all branches. We stopped our experiments after 50 iterations as SnarkFuzzer did not reach any new branches over the previous 10 prior iterations. Even with this, we note that after roughly 30 iterations we see the breadth of our coverage begin to taper off. A graph can be found in Fig. 3.

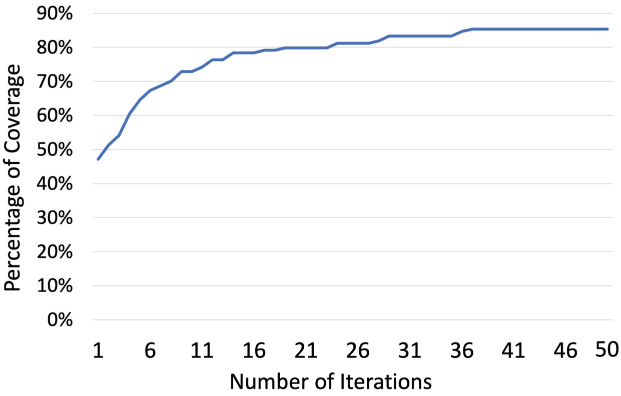


Fig. 3. Percentage (breadth) of library coverage after a given number of iterations

Performance of Value Model and Value Monitor. The primary effects on overall performance are our Value Model and Value Monitor. Table 4 shows the runtime with different matrix sizes. We expect the size of our matrices to be one of the primary influences on the runtime of these components.

Recall that the computation of the proving key requires calculations with the circuit. Therefore, a larger R1CS matrix will result in longer runtime in our Value Model. At the same time, the shape of the matrix will also affect its runtime. As we can see in Table 4, a matrix of size 20 × 45 takes less time than one of size 30 × 30 or 45 × 20, even though they all contain 900 integers. In fact, the number of variables (columns) plays a greater role than the number of

Table 4. Runtime of the Value Model and Value Monitor to process an R1CS matrix of varying size. For example, 10 × 30 is a matrix with 10 variables and 30 constraints.

Matrix Setting		Processing Time (in second)	
Matrix Size	Items in Matrix	Value Model	Value Monitor
10 × 10	100	23.33	29.21
10 × 30	300	25.96	33.09
30 × 10		32.78	34.21
20 × 45	900	29.52	43.99
30 × 30		33.01	46.30
45 × 20		40.14	47.68
45 × 60	2700	41.29	81.69
60 × 45		46.03	80.21
100 × 1500	150000	200.84	223.49
100 × 5000	500000	580.97	344.83

constraints (rows) in Value Model runtime, since this plays a larger role in the size of the proving and verification keys.

Similarly, the Value Monitor requires more time to watch variables in a program with a larger matrix. However, the reason for this is different. Since the runtime of Valgrind depends on the space complexity of a variable, the matrix shape will not have an effect. Instead, the Value Monitor spends more time monitoring a program with a larger matrix, but it takes similar time for programs with the same number of integers but different shapes. For example, each matrix of size 20 × 45, 30 × 30, and 45 × 20 has 900 integers, and the Value Monitor takes around 46 s to run, even though they have a different shape.

Discussion of Total Runtime. Our experiments only provide an upper bound on the worst case runtime for SnarkFuzzer. We did not perform any runtime optimizations, and in all of our experiments, we used the slowest options or configurations that we knew of. While SnarkFuzzer does require extra runtime in the Value Monitor and Value Checker for each individual input seed, in comparison to a traditional fuzzer, which might spend hundreds or thousands of core hours to produce inputs, the framework and Branch Model designs allowSnarkFuzzer to find issues with a relatively small number of generated input seeds. In fact, most errors that SnarkFuzzer detected (see Sect. 6) were found with less than ten seeds. Therefore, we view performance to be acceptable for real world usage.

Table 5. Summary of the current potential errors and inconsistencies, and previous vulnerabilities, that `SNARKProbe` is able to catch and locate.

Error/Vulnerability	Type	Affected Component	Program	Found by
Potentially Locatable in Current Libraries				
Incorrect manual flattening by developer	Logic Error	R1CS Circuit	All libraries	ConstraintChecker
Multiple gadget misuse	Logic Error	R1CS Circuit	All libraries	ConstraintChecker
Found in Current Libraries				
Incorrect bit/comparison gadget implementation	Logic Error	R1CS Circuit	libsnark	ConstraintChecker
Mismatch in circuit generation and usage	Logic Error	R1CS Circuit	All libraries	ConstraintChecker
Groth16 pre-pairing computation in Setup	Logic Error	Groth16 Protocol	bellman/arkworks	SnarkFuzzer
Inconsistent QAP extension index usage	Logic Error	PGHR13 Protocol	libsnark	SnarkFuzzer
Toxic waste not safely destroyed	Logic Error	PGHR13 Protocol	playsnark	SnarkFuzzer
Program out of memory with large circuit	Software Error	-	libsnark	SnarkFuzzer
Found in Previous Versions of Libraries				
CVE-2019-7167	Logic Error	PGHR13 Protocol	libsnark (2018)	SnarkFuzzer

6 Error Catching Evaluation

Our second set of evaluations demonstrates `SNARKProbe`'s ability to *automatically* catch different types of errors in zkSNARK libraries. We test `SNARKProbe` on `libsnark`, `Bellman`, `arkworks`, and `playsnark`, as well as on the circuit generator `Circom`. Additionally, we show how `SNARKProbe` would have caught a previous CVE in the zkSNARK component of Zcash, a popular privacy-preserving cryptocurrency. In total, `SNARKProbe` detected seven inconsistencies in current zkSNARK libraries, successfully *automatically* located the previous vulnerability, and could potentially detect two additional types of errors. We summarize our results in Table 5.

6.1 Potentially Locatable in Current Libraries

Incorrect Flattening. There is always a possibility that a developer may generate an incorrect R1CS matrix for their circuit, since this process is often done by hand. While we did not find any examples of this in the few applications we tested, consider the following scenario. Take a statement equation like $x^3 + x + 5 = y$ which the developer will typically flatten into `R1CS_gates` with private input $x = 3$ and public input $y = 35$. This set of equations is then equal to $x^3 + x + 5 = 35$. However, the developer *may* incorrectly flatten the statement equation into `R1CS_gates_2`.

$$\texttt{R1CS\_gates} = \begin{cases} x * x = w_1 \\ w_1 * x = w_2 \\ w_2 + x = w_3 \\ w_3 + 5 = y \end{cases} \qquad \texttt{R1CS\_gates\_2} := \begin{cases} x * x = w_1 \\ w_1 * x = w_2 \\ w_2 + 2 * x = w_3 \\ w_3 + 2 = y \end{cases}$$

While `R1CS_gates_2` still satisfies the private input $x = 3$ and public input $y = 35$, causing the underlying library to output an acceptable proof, this flattening does not in fact equal the statement equation $x^3 + x + 5 = y$. Although the statement equation and `R1CS_gates_2` satisfy the witness, and their domains have the same upper and lower bounds, they have a different output range, which our Constraint Checker will catch and flag as an error.

Combining Multiple Gadgets. Recall that a gadget is an abstraction that many zkSNARK libraries contain to make developing proofs easier for developers. Multiple gadgets may also be combined to create a single proof. Gadgets are generally created and provided independently, which means that one must combine them with care. For example, say a developer uses the gadgets *greater_than* and *equal_to* to create a proof that a prover has a value which is greater than or equal to a certain number. For this proof, only one input should be used for both gadgets. However, a naive developer may create a proof that has an individual input for each gadget instead of a single input for the entire circuit. In this case, a malicious prover can create a separate program with the same circuit and use two inputs with different values to satisfy each gadget and hence the entire circuit (when in reality these two inputs need to be identical). In this case, our Constraint Checker can detect the inconsistency between the R1CS matrix and statement equations to find potential issues with using multiple gadgets incorrectly.

6.2 Found in Current Libraries

Defective Gadgets. Gadgets may be implemented incorrectly, causing inconsistencies between the original (desired) statement equation and the R1CS gates generated. Our Constraint Checker is able to find these inconsistencies in its checks for equivalence, regardless of whether a gadget was used or not.

The `comparison_gadget` in `libsnark` uses bit shifting to compare two variables A and B by calculating $2^n + B - A$ and then counting the number of 0 s and 1 s in the resulting bit array (the corresponding R1CS constraints are shown in Appendix C). If the developer incorrectly specifies a bit length smaller than the size of A or B, the `comparison_gadget` will not generate the correct R1CS, which may result in a "fake proof"[9]. `libsnark` will not block this behavior or raise any warning messages. Consider a developer that uses the `comparison_gadget` to generate the statement equation $x < 60$ where $x \in \mathbb{F}_p$. The R1CS should then only have a valid solution in $[0, 60)$. However, if an incorrect bit length is provided, we found that all numbers in the range $(21888242871839275222246405745257275088548364400416034343698204186575808494653, p)$ still satisfy this incorrect R1CS, allowing a dishonest prover to generate a valid proof without knowledge of a correct secret value. After our tool flagged this potential issue, on further manual investigation we did see that the `libsnark` source code contains a comment about using the correct bit size, though this could go unnoticed by a novice user or be abused by a malicious one to forge proofs.

Mismatch Between Circuit Generation and Usage. In addition to end-to-end libraries, there are a growing number of circuit generators available to assist developers in constructing R1CS/circuits for proof generation, with `Circom` [41] being a widespread example[10]. `Circom` generates R1CS files that can then be utilized by libraries such as `snarkjs` [17], `wasmsnark` [18], and `rapidSnark` [16]. However, during our evaluation process, our Constraint Checker identified a crucial potential issue: the absence of attribute cross-checking between the circuit generator and formal proof generation library.

While the circuit generator may produce valid R1CS that accurately represents the prover's original statement, the proof generation library might, for example, employ a different elliptic curve and finite field to generate the proof by default. Consequently, when operating under distinct elliptic curve and finite field settings, the circuit no longer maintains equivalence with the original statement. This discrepancy can be problematic both for a developer and a verifier who are unaware that a circuit can yield different representations under different elliptic curve configurations. In such cases, the prover can produce a seemingly "valid" proof for the verifier without knowledge of the secret value.

[9] One that will verify even though the prover does not actually have a valid witness.

[10] We note that `Circom` is just an example that we tested, and that the same should hold true for other circuit generators as well.

Groth16 Pre-pairing. The Groth16 protocol produces $P = g^{\alpha}$ and $Q = h^{\beta}$ as part of Setup, which are subsequently used in Verify to calculate the pairing $e(P, Q)$. However, our Value Checker found that `Bellman` stores the pairing $e(P, Q)$ as part of the verification key instead of P and Q. Pre-computing and directly providing $e(P, Q)$ is an optimization that helps make verification more efficient (as pairings are quite slow to calculate). As this is a well-known optimization, we can confidently say that this will not cause any sort of vulnerability, but we err on the side of caution and flag all inconsistencies between the source code and protocol for further review since these can greatly increase the risk of actual vulnerabilities.

Pinocchio Protocol Inconsistencies. In the Pinocchio key generator, there exist values $\overrightarrow{A}$, $\overrightarrow{B}$, and $\overrightarrow{C}$ that need to be extended via $A_{m+1} = B_{m+2} = C_{m+3}$ and $A_{m+2} = A_{m+3} = B_{m+1} = B_{m+3} = C_{m+1} = C_{m+2}$ per the specification [27]. However, our Value Checker found that `libsnark` only extended $\overrightarrow{A}$, $\overrightarrow{B}$, and $\overrightarrow{C}$ via $A_{m+1} = B_{m+1} = C_{m+1}$ as an optimization for space complexity. After a review of the `libsnark` source code and protocol, we believe this inconsistency will not lead to a security vulnerability given how the values are used in practice. However, this instance is more subtle and less well-known than the previous one, so while it might not be exploitable, we believe inconsistencies of this nature are still valuable to flag for further review as this might not always be the case.

Toxic Waste. As part of the zkSNARK setup process, a set of private parameters informally referred to as "toxic waste" are generated. This toxic waste must be destroyed after the setup process, as possession of it allows one to forge proofs. However, our SnarkFuzzer identified an actual implementation error in a zkSNARK library called `playsnark`, whereby toxic waste is not properly destroyed during the setup phase. `playsnark` [5] serves as a learning playground for proof systems, including Pinocchio and Groth16. While `playsnark` makes no claims that it should be used for security-sensitive applications, we feel that this example still exemplifies valuable use cases for `SNARKProbe`. First, it can indeed catch errors that would be exploitable in real applications. And second, while libraries like `Bellman` and `libsnark` are professionally audited, this will likely not always be the case for all zkSNARK code in production usage in the future (as we have seen in many other domains). Tooling such as `SNARKProbe` can be an invaluable resource for non-experts and small organizations, enabling them to assess the security posture of unaudited zkSNARK libraries, audit their own code throughout the development process, and facilitating informed decisions regarding future security implementations.

`libsnark` **Out of Memory.** SnarkFuzzer detects a corner-case software error in `libsnark`. When our Input Generator tried to produce a large R1CS matrix with tens of thousands of variables and constraints, we discovered that `libsnark` cannot compile and produce a zkSNARK program with such a large R1CS matrix without throwing a segmentation fault or memory error. Intuitively, this occurs because the input matrix we generated was quite dense, i.e., it had a large number of both variables and constraints, as opposed to being quite sparse like most typical R1CS matrices. `libsnark`'s R1CS storage format is clearly not setup to handle such cases. In general, this would not impact the typical usage of the `libsnark`, as our fuzzer is designed to look for extreme cases like this that might not happen with a typical real world proof statement.

We take this example as a chance to take a step back and revisit our comparison of SnarkFuzzer to a traditional, more generic fuzzer like AFL. As generic memory and software errors such as this have nothing to do with the actual cryptography and zkSNARK logic, traditional fuzzers are also able to detect them. However, it is likely to take a generic fuzzer significantly longer (and many more inputs) to catch such errors, as they lack the context to understand what an "extreme" input means in this instance, and, as we have seen, it is not easy to instrument a fuzzer with this context. Additionally, all previous instances that we discussed would not be detectable by a generic fuzzer, as they are specifically cryptographic logic errors.

6.3 Found in Previous Versions of Libraries

CVE-2019-7167 Problem. Finally, we discuss how `SNARKProbe` (specifically SnarkFuzzer) is able to detect real world vulnerabilities by demonstrating its ability to pinpoint a previous CVE. CVE-2019-7167 [34] was found in `libsnark` through a manual code audit, as part of its usage in the popular privacy-preserving cryptocurrency Zcash [19] (and has since been patched). At a high level, this vulnerability produced a bypass element in the key generation that damaged the soundness of the zkSNARK proof system. To fix this vulnerability, the value of $pk_A^{'}$ must be replaced from $\{A_i(\tau)\alpha_A\rho_A\mathcal{P}\}_{i=0}^{m+3}$ to $\{A_i(\tau)\alpha_A\rho_A\mathcal{P}\}_{i=n+1}^{m+3}$, with all other values the same.

To test this, we downloaded a version of `libsnark` from 2018 (Commit hash: bf2146b). When running SnarkFuzzer, the Value Checker *automatically* found this inconsistency and vulnerability in key generation. Additionally, while `libsnark` has since fixed this issue, it did not directly modify the structure of $pk_A^{'}$ in order to keep consistency with the structure of the other proving keys. Therefore, $pk_A^{'}$ still has size $m + 3$ instead of $(m + 3) - (n + 1)$. `libsnark` just added an extra handler to reformat the $pk_A^{'}$ size in the prover, and while this does not appear to have caused any additional issues, it is unclear if this could result in other corner case issues.

7 Conclusion

In this paper we present SNARKProbe, a framework that can automatically and systematically check for potential software errors and cryptographic logic errors, as well as for consistency in the implemented proof statement, in R1CS-based zkSNARK libraries, with little manual input from the user. We demonstrated SNARKProbe's design flexibility by implementing it for multiple different zkSNARK protocols and libraries. In addition, we performed extensive performance evaluations, as well as tested its ability to detect potential errors and inconsistencies. We believe SNARKProbe is a step in the right direction for ensuring the correctness of existing zkSNARK implementations and applications.

Acknowledgments. This work was supported by NSF grant CNS-2047991.

A Additional Information for Constraint Checker

A.1 An optimization for the SMT solver

The SMT solver may take a while to solve some complicated equations. To reduce the processing time in the equality check, we introduce optimizations and simplify some equations in the R1CS matrix. For example, many libsnark gadgets use bit shifting, which produces the equation $(x*(1+(p-1)*x)) \bmod p = 0$ in R1CS gates. This equation represents $x = 0$ or $x = 1$. For our optimization, if the Constraint Checker detects such a relation, it will be replaced with $0 \leq x \wedge x \leq 1$ where $x \in \mathbb{F}$. These are logically equivalent, but our replacement is much easier for the SMT solver to work with. A real world gadget like the comparison_gadget in libsnark has 16 constraints, and there are 11 constraints representing $x = 0$ or $x = 1$ (see Appendix C).

A.2 Protocol of the Constraint Checker by the SMT Solver

```
(a) Parameters.

A set of statement equations $E_1$ in size $p$ where equation $e_1^i \in E_1$ and $i \in [1, p]$.

A set of equations $E_2$ converted from R1CS gates in size $q$ where $e_2^i \in E_2$ and $i \in [1, q]$.

$m$ private variables $x_1^i$ in $E_1$ where $i \in [1, m]$ and $n$ private variables $x_2^i$ in $E_2$ where $i \in [1, n]$ and $n \geq m$.
A set of
auxiliary input $v_x^i$.

$k$ public variables $y_1^i$ in $E_1$ and $k$ public variables $y_2^i$ in $E_2$ where $i \in [1, k]$. A set of primary input $v_y^i$.

A SMT Pro Solver $S$ and Optimizer $O$ with modulus $P$.
$0 \leq x_1 < P$, $0 \leq x_2 < P$, $0 \leq y_1 < P$, and $0 \leq y_2 < P$ without optimization.
```

```
(b) Function valid check.
INPUTS: $E_1$, $E_2$, $S$.
OUTPUTS: Boolean result of function validation.

 1. Check and solve the R1CS gates:
    Set a solvers $S_1 \in S$.
    Add $e_1^i \in E_1$ where $i \in [1, p]$ and range of $x_1^i$, $x_2^i$ to $S_1$.
    Solve $S_1$ and get the set of private variables solution as $M_{1_x}^i$ and set of public variables solution as $M_{1_y}^i$.
 2. Check and solve the statement equations:
    Set a solvers $S_2 \in S$.
    Add $e_2^i \in E_1$ where $i \in [1, q]$ and range of $y_1^i$, $y_2^i$ to $S_2$.
    Solve $S_2$ and get the set of private variables solution as $M_{2_x}^i$ and set of public variables solution as $M_{2_y}^i$.
 3. Output $v_x^i \in M_{1_x}^i$ AND $v_x^i \in M_{2_x}^i$ AND $v_y^i \in M_{1_y}^i$ AND $v_y^i \in M_{2_y}^i$.
```

```
(c) Equivalent comparison.
INPUTS: $E_1$, $E_2$, $S$.
OUTPUTS: Boolean result of function equality.
Set two solvers $S_1$ and $S_2 \in S$.

 1. Comparing with public variables:
    Add $e_1^i \in E_1$ where $i \in [1, p]$ and range of $x_1^i$, $y_1^i$ to $S_1$.
    Add $e_2^i \in E_2$ where $i \in [1, q]$ and range of $x_2^i$, $y_2^i$ to $S_1$.
    Add $x_1^i == x_2^i$ where $i \in [1, m]$ to $S_1$.
    Add $y_1^i \neq y_2^i$ where $i \in [1, k]$ to $S_1$.
    Check if $S_1$ is SAT, UNSAT, or UNSURE as $c_1$.

 2. Comparing with Boolean variables:
    Set $b_1$ and $b_2 \in$ BOOLEAN
    Add $b_1 = AND(E_1)$ to $S_2$.
    Add $b_2 = AND(E_2)$ to $S_2$.
    Add $x_1^i == x_2^i$ where $i \in [1, m]$ to $S_2$.
    Add $y_1^i == y_2^i$ where $i \in [1, k]$ to $S_2$.
    Add $b_1 \neq b_2$ to $S_2$.
    Check if $S_2$ is SAT, UNSAT, or UNSURE as $c_2$.

 3. Output ($c_1 ==$ UNSAT AND $c_2 ==$ UNSAT).
```

```
(d) Domain comparison.
INPUTS: $E_1$, $E_2$, $S$, $O$.
OUTPUTS: Boolean result of domain comparison.

 1. Comparing upper and lower bond:
    Add $e_1^i \in E_1$ where $i \in [1, p]$ and range of $x_1^i$, $x_2^i$ to $O$.
    Add $e_2^i \in E_1$ where $i \in [1, q]$ and range of $y_1^i$, $y_2^i$ to $O$.
    Solve the upper bond $u_1$ and lower bond $l_1$ from $O$
    Solve the upper bond $u_2$ and lower bond $l_2$ from $O$

 2. Check if R1CS is valid outside statement domain:
    Add NOT(AND(range of $x_1^i$, $x_2^i$)) to $S$.
    Add $e_2^i \in E_1$ where $i \in [1, q]$ and range of $x_2^i$, $y_2^i$ to $S$.
    Check if $S$ is SAT, UNSAT, or UNSURE as $c$.

 3. Output ($c ==$ UNSAT AND $u_1 == u_2$ AND $l_1 == l_2$).
```

A.3 Example Constraint Checker

For the interested reader, we provide an example of our Constraint Checker to show how a developer can run the tool. More examples can be found in our code.

```python
x = z3.Int("x")
## Indicate the public variables and private variables
allocate = ["x"]
variables = [x]
## Indicate the statement of a proof
statement = [x < 60]
## Provide the snark program to automatically extract the R1CS matrix
## Developer can also manually provide the R1CS matrix
path = os.path.join(currentdir, "range")
## Call the functions in ConstraintChecker to evaluate the correctness
fcmp = FunctionComparison(path)
```

```
fcmp.allocate(allocate)
fcmp.set_input_sizes(0)
fcmp.addVariables(variables)
fcmp.addStatement(statement)
fcmp.addGadget(gadget1.comparison_gadget("max")) ## Optional
fcmp.addRange(x, z3.And(x < 60))
fcmp.runComparisonTests()
```

A.4 Domain Gap Issues

The same upper bound and lower bound does not guarantee that statement equations and R1CS gates have the same range. Figure 4c and 4d shows an example of an unmatched domain with the same upper bound and lower bound.

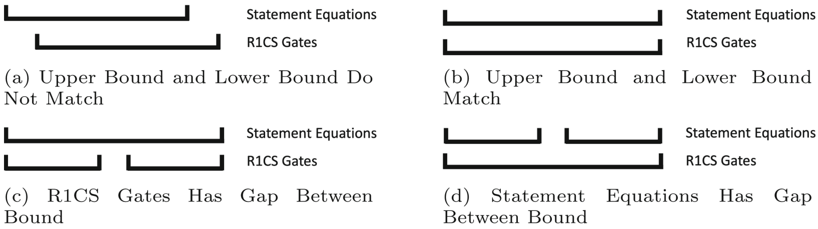
Domain Gap in Statement Equations

The ground truth domain of statement equations is known and provided by the user (which means we know the domain gap in the statement equations), the Constraint Checker can use the SMT solver to find if the R1CS gates have solution(s) outside the domain of statement equations (e.g. in the gap). If the SMT solver returns `SAT`, then the domains' of the statement equations and R1CS gates do not match and thus they are not equivalent. This example corresponds to Fig. 4d.

Domain Gap in R1CS Gates

Since the domain of the R1CS gates is unknown, the Constraint Checker cannot use the SMT solver to detect an unmatched domain as in the previous example. However, in this instance, the prover likely will not be able to generate a proof for a secret value in the gap because these secret values are not in the domain of R1CS Gates even though they are valid solutions to the original statement equations. That is, the set of allowable input values for the implemented R1CS matrix is a subset of the set of input values for the desired proof statement. Therefore, while this leads to undesirable behavior as the prover cannot generate proofs for the entire valid input range, it does not lead to any exploits or "fake" proofs (i.e., the ability to generate a valid proof without knowledge of the secret value). This example corresponds to Fig. 4c.

We believe this type of gap is very rare, but we still try to find this issue by producing a set of uniform random numbers as input for statement equations and R1CS gates. If the R1CS gates do not have a solution but the statement equations do have a valid solution for an input number, then there is a gap in the statement equations, which is not equivalent to the statement equations' domain. This test does not guarantee finding an existing domain gap, but a larger set of numbers has higher confidence in finding any existing gaps.



(a) Upper Bound and Lower Bound Do Not Match

(b) Upper Bound and Lower Bound Match

(c) R1CS Gates Has Gap Between Bound

(d) Statement Equations Has Gap Between Bound

Fig. 4. Examples of Domain Comparison

B Additional Information for SnarkFuzzer

B.1 Example Ideal Model Files

See Figs. 5 and 6.

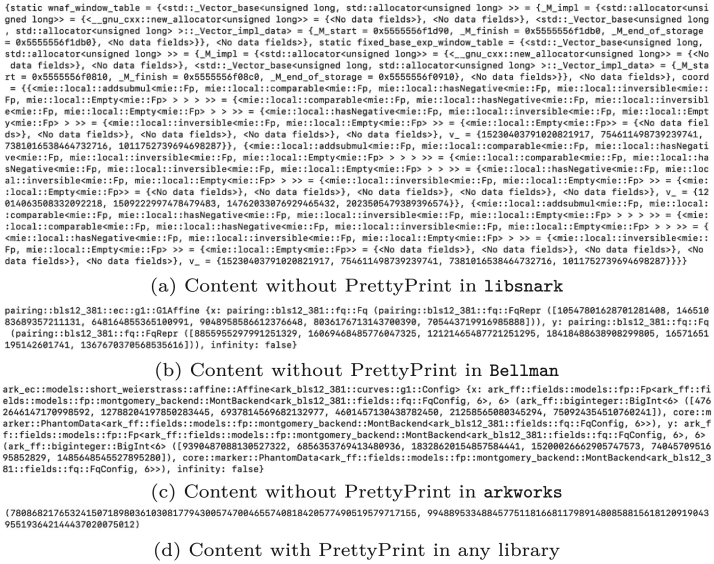
```
Relative Path: knowledge_commitment/kc_multiexp.tcc
FUNCTION,2,99,127,kc_batch_exp_internal,IMPORTANT
CONDITION,4,117,124,2,IMPORTANT
CONDITION,5,119,123,2,IMPORTANT
RETURN,2,126,126,2,IMPORTANT
```

Fig. 5. Example file for the ideal branch file

```
tau,Fr,t,zk_proof_systems/.../r1cs_ppzksnark.tcc,252
rhoA,Fr,rA,zk_proof_systems/.../r1cs_ppzksnark.tcc,300
rhoB,Fr,rB,zk_proof_systems/.../r1cs_ppzksnark.tcc,301
```

Fig. 6. Example file for the ideal value file

B.2 PrettyPrint Example



```
{static wnaf_window_table = {<std::_Vector_base<unsigned long, std::allocator<unsigned long> >> = {_M_impl = {<std::allocator<unsi
gned long>> = {<__gnu_cxx::new_allocator<unsigned long>> = {<No data fields>}, <No data fields>}, <std::_Vector_base<unsigned long
, std::allocator<unsigned long> >::_Vector_impl_data> = {_M_start = 0x5555556f1d90, _M_finish = 0x5555556f1db0, _M_end_of_storage
= 0x5555556f1db0}, <No data fields>}}, <No data fields>}, static fixed_base_exp_window_table = {<std::_Vector_base<unsigned long,
std::allocator<unsigned long> >> = {_M_impl = {<std::allocator<unsigned long>> = {<__gnu_cxx::new_allocator<unsigned long>> = {<No
 data fields>}, <No data fields>}, <std::_Vector_base<unsigned long, std::allocator<unsigned long> >::_Vector_impl_data> = {_M_sta
rt = 0x5555556f0810, _M_finish = 0x5555556f08c0, _M_end_of_storage = 0x5555556f0910}, <No data fields>}}, <No data fields>}, coord
 = {{<mie::local::addsubmul<mie::Fp, mie::local::comparable<mie::Fp, mie::local::hasNegative<mie::Fp, mie::local::inversible<mie::
Fp, mie::local::Empty<mie::Fp> > > > >> = {<mie::local::comparable<mie::Fp, mie::local::hasNegative<mie::Fp, mie::local::inversibl
e<mie::Fp, mie::local::Empty<mie::Fp> > > >> = {<mie::local::hasNegative<mie::Fp, mie::local::inversible<mie::Fp, mie::local::Empt
y<mie::Fp> > >> = {<mie::local::inversible<mie::Fp, mie::local::Empty<mie::Fp> >> = {<mie::local::Empty<mie::Fp>> = {<No data fiel
ds>}, <No data fields>}, <No data fields>}, <No data fields>}, <No data fields>}, v_ = {15230403791020821917, 754611498739239741,
7381016538464732716, 1011752739694698287}}, {<mie::local::addsubmul<mie::Fp, mie::local::comparable<mie::Fp, mie::local::hasNegati
ve<mie::Fp, mie::local::inversible<mie::Fp, mie::local::Empty<mie::Fp> > > > >> = {<mie::local::comparable<mie::Fp, mie::local::ha
sNegative<mie::Fp, mie::local::inversible<mie::Fp, mie::local::Empty<mie::Fp> > > >> = {<mie::local::hasNegative<mie::Fp, mie::loc
al::inversible<mie::Fp, mie::local::Empty<mie::Fp> > >> = {<mie::local::inversible<mie::Fp, mie::local::Empty<mie::Fp> >> = {<mie:
:local::Empty<mie::Fp>> = {<No data fields>}, <No data fields>}, <No data fields>}, <No data fields>}, <No data fields>}, v_ = {12
014063508332092218, 1509222997478479483, 14762033076929465432, 2023505479389396574}}, {<mie::local::addsubmul<mie::Fp, mie::local:
:comparable<mie::Fp, mie::local::hasNegative<mie::Fp, mie::local::inversible<mie::Fp, mie::local::Empty<mie::Fp> > > > >> = {<mie:
:local::comparable<mie::Fp, mie::local::hasNegative<mie::Fp, mie::local::inversible<mie::Fp, mie::local::Empty<mie::Fp> > > >> = {
<mie::local::hasNegative<mie::Fp, mie::local::inversible<mie::Fp, mie::local::Empty<mie::Fp> > >> = {<mie::local::inversible<mie::
Fp, mie::local::Empty<mie::Fp> >> = {<mie::local::Empty<mie::Fp>> = {<No data fields>}, <No data fields>}, <No data fields>}, <No
data fields>}, <No data fields>}, v_ = {15230403791020821917, 754611498739239741, 7381016538464732716, 1011752739694698287}}}}
```

(a) Content without PrettyPrint in libsnark

```
pairing::bls12_381::ec::g1::G1Affine {x: pairing::bls12_381::fq::Fq (pairing::bls12_381::fq::FqRepr ([10547801628701281408, 146510
83689357211131, 648164855365100991, 9048958586612376648, 8036176713143700390, 7054437199169858888])), y: pairing::bls12_381::fq::Fq
 (pairing::bls12_381::fq::FqRepr ([8855955297991251329, 16069468485776047325, 12121465487721251295, 18418488638908299805, 16571651
195142601741, 1367670370568535616])), infinity: false}
```

(b) Content without PrettyPrint in Bellman

```
ark_ec::models::short_weierstrass::affine::Affine<ark_bls12_381::curves::g1::Config> {x: ark_ff::fields::models::fp::Fp<ark_ff::fi
elds::models::fp::montgomery_backend::MontBackend<ark_bls12_381::fields::fq::FqConfig, 6>, 6> (ark_ff::biginteger::BigInt<6> ([476
2646147170998592, 12788204197850283445, 6937814569682132977, 4601457130438782450, 21258565080345294, 750924354510760241]), core::m
arker::PhantomData<ark_ff::fields::models::fp::montgomery_backend::MontBackend<ark_bls12_381::fields::fq::FqConfig, 6>>), y: ark_f
f::fields::models::fp::Fp<ark_ff::fields::models::fp::montgomery_backend::MontBackend<ark_bls12_381::fields::fq::FqConfig, 6>, 6>
(ark_ff::biginteger::BigInt<6> ([9390487088130527322, 6856353769413480936, 18328620154857584441, 15200026662905747573, 74045709516
95852829, 1485648545527895280]), core::marker::PhantomData<ark_ff::fields::models::fp::montgomery_backend::MontBackend<ark_bls12_3
81::fields::fq::FqConfig, 6>>), infinity: false}
```

(c) Content without PrettyPrint in arkworks

```
(780868217653241507189803610308177943005747004655740818420577490519579717155, 9948895334884577511816681179891480858815618120919043
955193642144437020075012)
```

(d) Content with PrettyPrint in any library

Fig. 7. Content of variable type G1

C Equations comparison_gadget Represents

$$
\begin{cases}
(w_{14} * (1 + (p-1) * w_{14})) \bmod p = 0 \\
(1024 + w_3 + 2 * w_4 + 4 * w_5 + 8 * w_6 + 16 * w_7 + 32 * \\
\qquad w_8 + 64 * w_9 + 128 * w_{10} + 256 * w_{11} + 512 * w_{12}) \\
\qquad \mathrm{mod} p = w_{13} \\
(w_3 * (1 + (p-1) * w_3)) \bmod p = 0 \\
(w_4 * (1 + (p-1) * w_4)) \bmod p = 0 \\
(w_5 * (1 + (p-1) * w_5)) \bmod p = 0 \\
(w_6 * (1 + (p-1) * w_6)) \bmod p = 0 \\
(w_7 * (1 + (p-1) * w_7)) \bmod p = 0 \\
(w_8 * (1 + (p-1) * w_8)) \bmod p = 0 \\
(w_9 * (1 + (p-1) * w_9)) \bmod p = 0 \\
(w_{10} * (1 + (p-1) * w_{10})) \bmod p = 0 \\
(w_{11} * (1 + (p-1) * w_{11})) \bmod p = 0 \\
(w_{12} * (1 + (p-1) * w_{12})) \bmod p = 0 \\
(1024 - 1 * w_1 + w_2) \bmod p = w_{13} \\
(w_{15} * (w_3 + w_4 + w_5 + w_6 + w_7 + w_8 + w_9 + w_{10} + \\
\qquad w_{11} + w_{12})) \bmod p = w_{14} \\
((1 + (p-1) * w_{14}) * (w_3 + w_4 + w_5 + w_6 + w_7 + w_8 \\
\qquad + w_9 + w_{10} + w_{11} + w_{12})) \bmod p = 0 \\
w_{14} \bmod p = 1
\end{cases}
$$

References

1. Fixing vulnerabilities in the zcash protocol (2016). https://electriccoin.co/blog/fixing-zcash-vulns/
2. Zcash counterfeiting vulnerability successfully remediated (2019). https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/
3. libsnark: a C++ library for zksnark proofs (2020). https://github.com/scipr-lab/libsnark
4. Nu4 cryptographic specification and implementation review (2020). https://research.nccgroup.com/wp-content/uploads/2020/09/NCC_Group_Zcash_ZCHX006_Report_2020-09-03_v2.0.pdf
5. Playsnark: a playground to learn proofs systems (2020). https://github.com/nikkolasg/playsnark
6. Dark forest (2022). https://blog.zkga.me/
7. Zero-knowledge rollups (2022). https://ethereum.org/en/developers/docs/scaling/zk-rollups/
8. Aleo (2023). https://www.aleo.org/
9. atheris, atheris: A coverage-guided, native python fuzzer (2023). https://github.com/google/atheris
10. bellman, a zk-SNARK library (2023). https://github.com/zkcrypto/bellman
11. Clang's source-based code coverage (2023). https://clang.llvm.org/docs/SourceBasedCodeCoverage.html
12. doxygen, doxygen (2023). https://github.com/doxygen/doxygen
13. gnark zk-SNARK library (2023). https://github.com/Consensys/gnark
14. libsnark: A rust implementation of the groth16 zkSNARK (2023). https://github.com/arkworks-rs/groth16
15. Project everest (2023). https://project-everest.github.io/
16. rapidsnark (2023). https://github.com/iden3/rapidsnark
17. snarkjs (2023). https://github.com/iden3/snarkjs
18. wasmsnark (2023). https://github.com/iden3/wasmsnark
19. Zcash (2023). https://z.cash/
20. zksnarks for the world (2023). https://research.protocol.ai/sites/snarks/
21. Aumasson, J.P., Romailler, Y.: Automated testing of crypto software using differential fuzzing. Black Hat USA (2017)
22. Bang, L., Rosner, N., Bultan, T.: Online synthesis of adaptive side-channel attacks based on noisy observations. In: IEEE EuroS&P (2018)
23. Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.: Extending SMT solvers to higher-order logic. In: CADE (2019)
24. Barbosa, M., et al.: SoK: computer-aided cryptography. In: IEEE S&P (2021)
25. Beck, G., Zinkus, M., Green, M.: Automating the development of chosen ciphertext attacks. In: USENIX Security (2020)
26. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. Cryptology ePrint Archive, Report 2013/507 (2013)
27. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von Neumann architecture. In: USENIX Security (2014)
28. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: ACM CCS (2017)
29. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as Markov chain. In: IEEE TSE (2017)

30. Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., Liu, W.: A systematic review of fuzzing techniques. Comput. Secur. **75**, 118–137 (2018)
31. Ethereum: Python implementation of ECC pairing and bn_128 and bls12_381 curve operations. Ethereum, December 2021. https://github.com/ethereum/py_ecc. Accessed Dec 2021
32. Faz-Hernández, A., Kwiatkowski, K.: Introducing CIRCL: An Advanced Cryptographic Library. Cloudflare, June 2019. https://github.com/cloudflare/circl. v1.2.0 Accessed June 2022
33. Fredrikson, M., Livshits, B.: Zø: An optimizing distributing zero-knowledge compiler. In: USENIX Security (2014)
34. Gabizon, A.: AuroraLight: improved prover efficiency and SRS size in a Sonic-like system. Cryptology ePrint Archive, Paper 2019/601 (2019). https://eprint.iacr.org/2019/601
35. Garman, C., Green, M., Kaptchuk, G., Miers, I., Rushanan, M.: Dancing on the lip of the volcano: chosen ciphertext attacks on apple {iMessage}. In: USENIX Security (2016)
36. Godefroid, P., Peleg, H., Singh, R.: Learn&Fuzz: machine learning for input fuzzing. In: ASE (2017)
37. Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In: FOCS (1986)
38. Google: syzkaller - kernel fuzzer (2017). https://github.com/google/syzkaller. Accessed July 2022
39. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_11
40. Householder, A.D., Foote, J.M.: Probability-based parameter selection for black-box fuzz testing. Technical report. Carnegie Mellon University, SEI (2012)
41. iden3: circom - Circuit Compiler for ZK Proving Systems (2023). https://github.com/iden3/circom. Accessed Aug 2022
42. Kosba, A.: xJsnark (2022). https://github.com/akosba/xjsnark. Accessed Aug 2022
43. Kupser, D., Mainka, C., Schwenk, J., Somorovsky, J.: How to break XML encryption-automatically. In: USENIX WOOT (2015)
44. o1 labs: snarky (2023). https://github.com/o1-labs/snarky. Accessed Aug 2022
45. Macgillivray, A., deBlanc Knowles, T.: Advancing a vision for privacy enhancing technologies (2022). https://www.whitehouse.gov/ostp/news-updates/2022/06/28/advancing-a-vision-for-privacy-enhancing-technologies/
46. Mouha, N., Raunak, M.S., Kuhn, D.R., Kacker, R.: Finding bugs in cryptographic hash function implementations. IEEE Trans. Reliab. **67**(3), 870–884 (2018)
47. Moura, L.D., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS (2008)
48. Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: ACM CCS (2015)
49. Nicas, J., Isaac, M., Frenkel, S.: Millions flock to telegram and signal as fears grow over big tech (2021). https://www.nytimes.com/2021/01/13/technology/telegram-signal-apps-big-tech.html
50. Nitulescu, A.: zk-snarks: a gentle introduction. Technical report (2020)
51. NSA: Patch critical cryptographic vulnerability in Microsoft Windows clients and servers (2020). https://media.defense.gov/2020/Jan/14/2002234275/-1/-1/0/CSA-WINDOWS-10-CRYPT-LIB-20190114.PDF
52. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: IEEE S&P (2013)

53. Pasareanu, C.S., Phan, Q.S., Malacaria, P.: Multi-run side-channel analysis using Symbolic Execution and Max-SMT. In: IEEE CSF (2016)
54. Phan, Q.S., Bang, L., Pasareanu, C.S., Malacaria, P., Bultan, T.: Synthesis of adaptive side-channel attacks. In: IEEE CSF (2017)
55. Protzenko, J., Beurdouche, B., Merigoux, D., Bhargavan, K.: Formally verified cryptographic web applications in WebAssembly. In: IEEE S&P (2019)
56. Rupprecht, D., Kohls, K., Holz, T., Pöpper, C.: Call me maybe: eavesdropping encrypted LTE calls with ReVoLTE. In: USENIX Security (2020)
57. Somorovsky, J.: Systematic fuzzing and testing of TLS libraries. In: ACM CCS (2016)
58. Walz, A., Sikora, A.: Exploiting dissent: towards fuzzing-based differential black-box testing of TLS implementations. In: IEEE TDSC (2017)
59. Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: ACM CCS (2013)
60. Zalewski, M.: American fuzzy lop (2016). https://github.com/mirrorer/afl. v2.52b Accessed July 2022