# Efficient Proofs of Software Exploitability for Real-world Processors

Matthew Green
mgreen@cs.jhu.edu
Johns Hopkins University
USA

Mathias Hall-Andersen
mathias@hall-andersen.dk
Aarhus University
Denmark

Eric Hennenfent
eric.hennenfent@trailofbits.com
Trail of Bits
USA

Gabriel Kaptchuk
kaptchuk@bu.edu
Boston University
USA

Benjamin Perez
benperez1227@gmail.com
Trail of Bits
USA

Gijs Van Laer
gijs.vanlaer@jhu.edu
Johns Hopkins University
USA

## ABSTRACT

We consider the problem of proving in zero-knowledge the existence of vulnerabilities in executables compiled to run on real-world processors. We demonstrate that it is practical to prove knowledge of real exploits for real-world processor architectures without the need for source code and without limiting our consideration to narrow vulnerability classes. To achieve this, we devise a novel circuit compiler and a toolchain that produces highly optimized, non-interactive zero-knowledge proofs for programs executed on the MSP430, an ISA commonly used in embedded hardware. Our toolchain employs a highly optimized circuit compiler and a number of novel optimizations to construct efficient proofs for program binaries. To demonstrate the capability of our system, we test our toolchain by constructing proofs for challenges in the Microcorruption capture the flag exercises.

## KEYWORDS

zero-knowledge proof, NIZK, Reverie, exploits, real-world processors, MSP430, KKW

## 1 INTRODUCTION

The proliferation of complex and critical software systems has given rise to the bug bounty paradigm, in which independent vulnerability research teams uncover and disclose ways to exploit deployed software in exchange for financial rewards. This process has resulted in the disclosure of several high-profile exploits in recent years [46], and hundreds of millions of dollars are awarded in bounties annually.

While bug bounty programs are invaluable to improving the security of software, they are plagued by issues of trust. Because vulnerability researchers and bug bounty program managers are not part of the same organization—and likely have no prior relationship—each side must trust that the other will fulfill their obligations honestly. Specifically, bug bounty program managers must trust that vulnerability research teams are not overselling their capabilities and have discovered a serious exploit. On the other hand, vulnerability research teams worry that those managing bug bounty programs will adaptively change the reward after disclosure of the exploit, claiming that the exploit does not meet some criteria.

Currently, vulnerability researchers and bug bounty program managers bridge this trust gap by having the vulnerability research team "prove" its knowledge of an exploit using a video recording. Concretely, the bug bounty program will challenge the vulnerability research team to perform an operation that should be impossible (*e.g.*, launching the calculator application) and visually record the program execution. These proofs lack soundness, as video can easily be manipulated and cannot prove that the runtime environment matches the one specified by the bug bounty program. As such, the state of the art still leaves significant trust gaps within the bug bounty ecosystem.

In this work, we design a toolchain that bridges this trust gap using cryptographically sound proofs of exploit. These proofs give a computational guarantee that the vulnerability research team can exploit the system within the specified runtime environment, and they cannot be manipulated or forged. To ensure that these proofs do not disclose anything else to the bug bounty program team, we employ zero-knowledge (ZK) [23, 24] proofs, a class of proof systems that reveals nothing to the verifier beyond the veracity of the statement. Access to ZK proofs of exploit would allow vulnerability researchers and bug bounty programs to negotiate rewards without requiring significant leaps of faith.

Designing efficient ZK proofs of exploit requires both overcoming significant engineering challenges and non-trivial theoretical contributions. While prior work [27, 28] has contemplated similar applications, their systems are limited to proving the existence of *potential vulnerabilities or bugs* in publicly available source code—falling short of meeting the needs of the vulnerability research market. In our work, we precisely model real processor architectures and runtime environments within the ZK protocol, allowing our proofs to reason directly about compiled binaries. Therefore, the proofs that our toolchain produces guarantee that the exploits will work on hardware. This level of fidelity is essential for allowing vulnerability research teams to precisely articulate and demonstrate their capabilities.

**Envisioned Workflows.** In order to illustrate the value of our techniques, consider three concrete ways that cryptographically sound proofs of exploit could be used:


*Proceedings on Privacy Enhancing Technologies 2023(1), 627–640*

https://doi.org/10.56553/popets-2023-0036

(1) A vulnerability research (VR) team responds to a public bug bounty by submitting their ZK proof of exploit. Once the sponsor has verified the proof, a reward amount is determined and put into escrow until the VR team submits the exploit.
(2) A VR team discovers a bug in a piece of software for which there is no bug bounty program. If the developers choose not to award a bounty after initial discussions, the VR team could post the ZK proof of exploit to a public website, informing users that their existing systems are at risk. Critically, this does not *reveal* the exploit to malicious actors who might want to use the exploit to attack live systems. We note that this would also put pressure on developers to issue a bounty and patch their software, as responsible users will likely transition away from their products.
(3) A VR team discovers a bug in a piece of legacy software which is no longer maintained, or is running on devices that cannot perform firmware updates. The VR team can post the proof of vulnerability to a public website, creating a highly trustworthy warning against using the legacy software. If using the legacy software is unavoidable, we note that users could crowdsource funds to hire the VR team to design and issue a patch.

We note that these are only potential examples, and proofs of exploit may be valuable in other workflows.

## 1.1 Contributions

In this work, we design the first end-to-end modular toolchain that facilitates the creation of ZK proofs of program exploitability.[1] The toolchain takes in two inputs: (1) a public compiled binary,[2] and (2) the prover's private input that exploits a vulnerability in that program. Given these inputs, it then produces a non-interactive zero-knowledge proof (NIZK) of correct execution. This is conducted by evaluating the binary as a RAM program using a Boolean processor circuit. While previous work has explored the evaluation of RAM machines using custom-built processors, our system employs real-world processor architectures; to make our system efficient, we introduce several novel processor-agnostic techniques that reduce the size of the resulting circuit. Specifically, we reduce the size of the circuit from $O(t \log(t))$ to $O(t)$, where $t$ is the number of processor cycles executed during program execution.

To evaluate the effectiveness of our toolchain, we produced ZK proofs of exploit for MSP430 binaries. First, we design a custom circuit implementation of the MSP430 processor that is optimized for ZK; this requires modeling system calls (syscalls) and complex addressing modes while minimizing the number of non-linear gates. Second, we provide the first public, generic implementation of the Katz, Kolesnikov, and Wang (KKW) "MPC-in-the-head" ZK protocol [33] and incorporate several significant improvements. Specifically, we show that the MPC-in-the-head with preprocessing paradigm that they propose can be modified to allow for optimized ring switching between Boolean and arithmetic representations, resulting in significantly more efficient proofs. Finally, we demonstrate the effectiveness of our approach by producing proofs of exploit for the Microcorruption CTF [26], a set of hacking challenges that run on an MSP430 processor and cover many common exploitation techniques such as buffer overflow, command injection, and ROP gadgets. The Microcorruption challenges also require bypassing mitigations such as address space layout randomization (ASLR), data execution protection (DEP), and stack canaries. Our toolchain can produce NIZK proofs about MSP430 programs at 216 instructions per second and 119 KB per instruction.[3]

**Limitations.** Our approach allows proofs about exploits that can be represented as a predicate over the processor states over a program's execution. This means that there are some classes of exploits about which we cannot provide proofs, like exploits that rely on microarchitectural bugs such as Spectre and Meltdown. Similarly, Row Hammer-style exploits cannot be expressed as such a predicate, as they require modeling physical properties of RAM. Accurately modeling these systems is challenging, independent of zero-knowledge proving; as such, these exploits are beyond the scope of this work. We note, however, that only the most sophisticated actors could successfully launch such an attack, and there are no documented cases of such exploits being used in the wild.

We note that our proofs do not attempt to conceal the running time of the exploit; the number of processor ticks required is included as a public part of the statement. This is a standard relaxation in prior work [10, 11, 13, 27], and given the trade-off of less efficient proofs, it is easy to "pad-out" the running time to conceal the trace length. Additionally, we note that any low-entropy probabilistic protections (*e.g.* ASLR) will always be vulnerable to computationally powerful adversaries, both for adversaries attacking live systems, *e.g.* using brute force, and for a prover generating a proof of exploit, *e.g.* grinding on random seed selection. This means that the meaning of a proof of exploit that overcomes low-entropy probabilistic defenses are nuanced: (1) when a proof is generated interactively and the processor randomness is sampled by the verifier, the proof implies that the prover has an exploit strategy that works on average, but may not always work; (2) when the proof is generated non-interactively, *i.e.* a computationally powerful prover may (invisibly) expend significant resources generating an accepting proof, the proof implies that there exists processor randomness such that the prover possesses a working exploit strategy.

Finally, we note that while our solution demonstrates the proofs of exploit are already practical, there remains more effort—both research and engineering—for the solution to be simple and easy to use. For example, vulnerability researchers must select the statement that they wish to prove carefully. Choosing the wrong statement could result in a proof that verifies but is semantically meaningless.

**Ethical Concerns.** Software exploits can be used to cause harm to people and organizations and there exist online markets where exploits are sold for nefarious purposes. As such, the techniques that we develop might also be used by individuals intent on causing harm. We note, however, that our techniques do not meaningfully

[1] Although prior work has explored the possibility of proving the existence of *bugs* in source code, our work addresses a fundamentally harder problem of demonstrating that a bug can be exploited into a full exploit. We carefully contrast these two approaches in Section 3.
[2] Our toolchain can naturally also operate from program source, which is compiled using a standard compiler.
[3] For hardware specifications, see Section 8

increase the capabilities of these communities; allowing hackers prove—with cryptographic soundness error—that they know an exploit only serves to make exploit markets more trustworthy and more easily monitored. Critically, our techniques do not make it easier for attackers to discover or exploit vulnerabilities or meaningfully increase a hacker's power to conduct blackmail.

# 2 TECHNICAL OVERVIEW

## 2.1 Background: Zero-Knowledge and Ben-Sasson et al.'s RAM Reduction

Zero-knowledge proofs of knowledge (ZK) [23, 24] allow a prover to convince a verifier that they hold a witness demonstrating that some public statement is a member of an NP language without revealing anything beyond the membership itself. ZK techniques are now concretely efficient [1, 5, 8, 12, 16–18, 22, 28, 32, 33, 52, 54, 55] and power a number of practical applications [9, 37, 49, 56]. For formal definitions of ZK proofs of knowledge, see [45].

Most research on ZK focuses on the case in which the statement is provided in a format amenable to efficient proving systems (*e.g.*, a circuit or algebraic relation). Therefore, most proof techniques now *require* that the relations have such a representation. This requirement can be unnatural and cumbersome, forcing implementers to translate a relation from its "natural" representation to the representation supported by the prover. This process frequently involves error-prone manual effort or the use of an immature circuit compiler [6, 35, 39, 51].

**RAM Reduction.** Ben-Sasson et al. [10, 11, 13] proposed an efficient circuit-based approach for proving the correct execution of RAM programs which has also been used by more recent works [21, 27, 29]. They represent the execution of the RAM program with two different traces. The first is the *execution-ordered trace*, wherein each step represents a single iteration of a processor circuit, including instruction bytes, a register file, and the alleged contents of memory being accessed. The second is the *memory-ordered trace*, containing the set of memory reads and writes sorted by address, with ties broken by the operation that was executed first. Proving that these traces represent an honest execution of the RAM program consists of the following:

(1) **Execution Trace Consistency.** For each step in the execution trace, the proof must demonstrate that the input and output states represent a valid transition. This is done using a circuit that represents the processor. The input to each evaluation of this circuit is a fixed number of values drawn from RAM, a register file, and other auxiliary data that may be useful in verifying correct execution. This circuit will output 1 if the circuit produces the same output as the real RAM program.
(2) **Memory Trace Consistency.** Each step in the trace involves reading and writing some values from RAM. Naïvely ensuring that these reads and writes are consistent with the previously executed instructions would require verifying the entire contents of RAM in each step. Instead, they maintain an address-ordered list called the memory trace, consisting of tuples of the form (step, operation, address, value), where step is a unique index in the execution trace, operation can either be read or write, and address is a location in memory [11]. The memory consistency circuit ensures that each read operation contains the same value as the most recent write operation to that address.
(3) **Permutation Check.** The two proofs above ensure that the execution trace is consistent with the processor circuit and that the operations in the memory trace are valid. However, we must still demonstrate that these traces are consistent with one another; that is, the values provided to the execution trace consistency circuit correspond to the elements verified using the memory trace consistency circuit. To ensure this consistency, we employ a permutation check that proves a one-to-one mapping between each read/write in the execution trace and some entry in the memory trace.

## 2.2 Formalizing Exploits

In order to produce cryptographically sound proofs of exploitability, we must have a formal NP language of which we can show a binary is a member. In our work, we are able to prove any exploit that is an arbitrary boolean predicate over the execution trace. Specifically, we can show that repeatedly applying the processor circuit to the processor state (for some public number of iterations) resulted in a processor state (or series of process states) that should have been impossible under honest execution. As such, we begin by designing circuit representations of real-world processors that are ZK friendly.

**MSP430.** In this work, we demonstrate the concrete feasibility of producing proofs of exploit for unaltered MSP430 binaries. MSP430 is a family of microprocessors commonly used in low-power environments. The version of the MSP430 ISA on which we focus has 27 instructions, including 12 double operand instructions (*e.g.* `MOV`, `ADD`, `AND`, `SUB`), 7 single operand instructions (*e.g.* `PUSH`, `CALL`), and 8 jump instructions (*e.g.* `JEQ`, `JNE`) [30, 36].

There are several significant obstacles to designing a circuit that implements the MSP430 instruction set architecture (ISA). MSP430 goes beyond a classic load/store architecture by incorporating 13 addressing modes. We augment our processor circuit using a set of memory hints in each step that provide the processor with the required information to complete the cycle's operation. The contents of the memory hints are interpreted based on the current instruction and are verified using the memory checker.

Given that the MSP430 is a small embedded processor it does not have an equivalent to system calls (syscalls) that are common in modern processors supported by full operating systems. Nevertheless, in some applications, including the Microcorruption CTFs, a library can introduce the equivalent of certain system calls. We take a similar approach as in the creators of the Microcorruption CTFs to add syscalls to the MSP430 ISA. We will give more details about this modeling in Section 4.2.

**Processor Predicates.** There are many predicates over the execution trace that are highly relevant to demonstrating exploitability. For example, one simple predicate would be that the final program counter (PC) in the trace is some particular challenge value; if an attacker can set the PC arbitrarily, they likely can execute arbitrary code. We also consider more complex predicates, like showing that a syscall was executed during the trace that should have been impossible (*e.g.* turning on the device's microphone). Predicates about syscalls can also be used to show privilege escalation, by showing that the `GETEUID` syscall returned the value 0. Selecting the

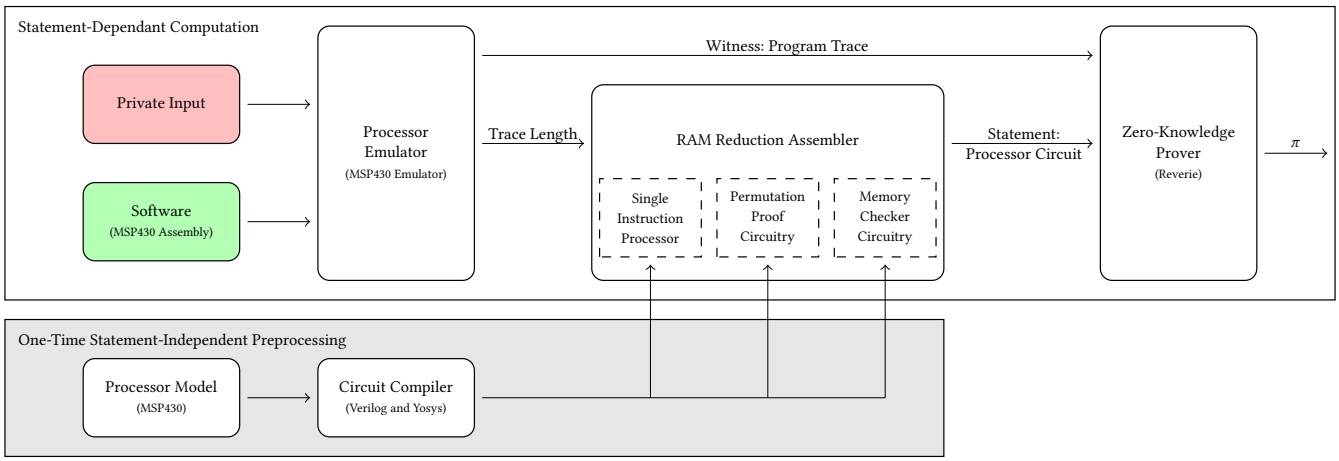


**Figure 1: A high level overview of our toolchain for producing efficient zero-knowledge proofs for RAM programs on real processors. (1) The process starts with a one-time preprocessing phase which compiles the processor model into building blocks which are later assembled into a complete circuit. The circuit compiler (which we instantiate using Verilog and Yosys) generates the circuit for evaluating a single instruction, and the circuitry required to perform the permutation proof and check memory correctness. (2) When the prover wishes to create a proof, they feed the software, represented as assembly in the appropriate ISA, and any private program inputs into the processor emulator. The processor emulator runs the program to its conclusion and outputs the execution trace. (3) Based on the length of the trace, the RAM Reduction Assembler takes the preprocessed circuit components and creates the completed circuit. (4) The program trace, produced by the processor emulator, and the completed circuit, produced by the RAM reduction assembler, into any zero-knowledge prover to produce the final proof. We include the instantiations we use for our proofs of vulnerability in parenthesis.**

right predicate—or set of predicates—is an exploit-specific task that can be done by either the vulnerability researcher (once they have found an exploit) or the bug bounty program when setting their bounties.

To support such predicates, we add syscall support to our processor circuit, making it the first ZK processor to include syscalls. When the program encounters a syscall, the processor freezes the registers and enables the finite state machine. The processor executes the syscall for an arbitrary number of steps until some exit condition is met (*e.g.*, for the `GETS` syscall, until the processor reads a maximum number of characters or encounters a null byte). The processor then unfreezes and continues execution. This allows syscalls to be unrolled on the fly without requiring significant, special-purpose circuitry.

## 2.3 Producing Efficient ZK Proofs of Exploit

With a formalization of exploits in hand, we develop a toolchain to produce proofs of exploit. An overview of our toolchain can be found in Figure 1, including a processor emulator, the RAM reduction assembler, and the ZK prover. The remaining task is to develop the necessary cryptographic optimizations such that the proofs of exploit that our toolchain produces are *efficient*.

**Notation.** We use $[b]$ for a share of a bit $b$, similarly we will use $[\![x]\!]$ for an arithmetic share of an element $x$ in an arithmetic ring.

**Reverie.** Our second main technical contribution in this work is Reverie, the first publicly available,[4] general use implementation of the KKW MPC-in-the-head ZK protocol [33]. Reverie is written in Rust and incorporates many optimizations to make it more efficient, including bit slicing, memory efficient representations of the circuit, and proof streaming. The prover can compute the root of a Merkle tree with 256 leaves in just 8 seconds, significantly faster than prior NIZK implementations (see Table 2 in Section 8).

Reverie also improves on KKW's initial protocol by including efficient ring switching based on edaBits [20]. To switch an element between rings, the prover generates shares of random elements in the two relevant rings during preprocessing. The prover then masks the value, reconstructs it in the clear, ring switches the public element, and removes the secret-shared mask.

For example, consider ring switching a value $v \in \mathbb{F}_{2^{32}}$ into an equivalent binary decomposition $(v_1, v_2, \ldots, v_{32}) \in \mathbb{F}_2^{32}$. The prover begins by generating random sharings of the values $r \in \mathbb{F}_{2^{32}}$ and $(r_1, r_2, \ldots, r_{32}) \in \mathbb{F}_2^{32}$ for the simulated players during the preprocessing, subject to the constraint $r = \sum_{i=1}^{32} r_i 2^i$. During online execution, the simulated parties publicly reconstruct the value $v + r$ and then decompose the public value into its binary representation $(v_1 + r_1), \ldots, (v_{32} + r_{32})$. The simulated parties then subtract their local shares of $r_1, \ldots, r_{32}$, resulting in a valid secret sharing of the values $(v_1, v_2, \ldots, v_{32}) \in \mathbb{F}_2^{32}$. This ring switching protocol is very efficient because generating verifiable, structured correlated randomness during preprocessing is very communication and computation efficient when using the KKW ZK protocol.

**Efficient Permutation Proof.** The RAM reduction outlined in Section 2.1 uses a routing network to implement the permutation proof between the execution trace and the memory trace. The routing network has asymptotic complexity $O(t \log(t))$, where $t$ is the trace length, and large constants. A more efficient permutation proof, first explored by [15, 41], shows that two secret lists $\{A_i\}_{i \in [\ell]}$ and $\{B_i\}_{i \in [\ell]}$ are permutations by sampling a random challenge $x \xleftarrow{\$} \mathbb{Z}_q$ and testing if

$$\prod_{i=1}^{\ell} (A_i - x) \stackrel{?}{=} \prod_{i=1}^{\ell} (B_i - x).$$

[4] https://github.com/trailofbits/reverie

To ensure that this test has negligible soundness error, it must be performed in a large field. However, our MSP430 processor operates over $\mathbb{F}_2$. Thus, the ring switching technique introduced above is vital to facilitating this permutation proof. Without access to an efficient ring switching technique, the test would have to be carried out in a small field with large soundness error, or the processor would need to operate over a large field, which would introduce high computational overhead. Concretely, the permutation proof costs just 380 AND gates and 2 multiplications for each element in the list.

**Evaluation.** We evaluate our toolchain by producing ZK proofs of exploitability for the Microcorruption Capture The Flag (CTF) exercises. Microcorruption CTF is a series of popular embedded device (MSP430) exploitation exercises that are freely available online. These exercises serve as a common entry point for individuals wishing to learn binary exploitation. Each challenge is named after a world city (see Table 1), and the exercises cover many common exploit techniques, such as heap and buffer overflows. Additionally, the processor implements important mitigation strategies, such as stack canaries, DEP, and ASLR. Thus, producing proofs of exploits for the Microcorruption CTF exercises demonstrates a wide variety of exploitation techniques, demonstrating the practicality of our approach.

The prover begins by initializing the processor emulator to a fresh state and loads the public binary. The prover then emulates the binary when run on the private input, which produces an execution trace containing the processor state for each step and a memory trace containing the memory operations for each step. This emulation process stops once the desired processor state is reached (*e.g.*, the processor makes a restricted syscall). The prover then assembles the unrolled circuit from the pre-compiled library of components based on the length of the traces. The assembled circuit is provided as the statement to the ZK prover, and the traces are provided as a witness. Note that the only requirement we make of the ZK prover is that it is capable of performing ring switching.

Concretely, in one second, our implementation can produce a NIZK of correct processor execution of 216 MSP430 instructions requiring 119 KB of communication per instruction.

## 3 RELATED WORK

**Modeling RAM Programs in ZK.** TinyRAM [11] and BubbleRAM [27] are two custom ISAs developed to maximize performance with existing ZK schemes. They both use a load/store architecture with fewer than 30 instructions and ensure that decoding each instruction is inexpensive within a ZK prover. Among such works are vRAM [57] which constructs verifiable computation with a universal trusted setup for the TinyRAM ISA. The aims of our work differs from those in the verifiable computation literature in a number of important ways: (1) the proof size is linear (in particular the verifier complexity is linear). (2) we aim for concretely efficient prover complexity by using only symmetric key operations as opposed, *e.g.* to pairings in vRAM. (3) our techniques do not rely on a trusted setup (universal or otherwise) (4) we target real-world architecture. Despite proving a much more complicated architecture the proving speed (emulated CPU cycles/second) in this work (for MSP430) is $\approx$ 5 times greater than vRAM (for TinyRAM). While Ben Sasson et al. [13] later modified TinyRAM to have a von Neumann architecture, BubbleRAM remains a Harvard architecture processor, which prevents it from reasoning about exploits that inject malicious code onto the stack or heap. As we discuss in the next subsection, the use of these custom ISAs limits the capabilities of a prover. For example, provers compile source code to the custom ISA, and source code is not available for many pieces of security critical software.

**Proofs of Exploitability.** In discussing prior work, we emphasize the difference between a vulnerability and an exploit. An exploit is maliciously crafted program input that produces unintended program behavior—or may even allow an attacker to affect the state of the computer beyond the program itself. A vulnerability, on the other hand, is a software weakness that could *potentially* be used in designing an exploit, for example an out-of-bounds memory write or a use-after-free bug. Vulnerabilities do not depend on architecture-specific constructs like the stack, heap, or mitigations such as ASLR, DEP, and pointer authentication codes (PAC). An exploit, however, is intrinsically linked to processor semantics. Therefore, it is not sufficient to reason only about source code when demonstrating the existence of an exploit.

Prior work on using ZK proofs for vulnerability disclosure [27, 28] has focused on manually annotating `C` code with assertions that a prover must demonstrate they can violate. This is accomplished by compiling the annotated code either directly to a circuit or to a custom ZK processor (*e.g.*, TinyRAM [11] or BubbleRAM/BubbleCache [27, 29]). While this approach is capable of proving many interesting vulnerabilities with extremely high efficiency, it has several drawbacks.

First, annotation of complex, real-world programs is time-consuming and error-prone. Source annotations cannot express many of the most commonly exploited classes of bugs [38, 48], and even the bugs theoretically detectable with annotations are difficult for programmers to find. Even if all these limitations could be overcome, this approach inherently requires access to source code, which is often not available.

Second, bugs in source do not always translate to exploits on a real processor. The example used by Heath and Kolesnikov [28] focuses on proving the existence of an out-of-bounds memory access—an operation many compilers will automatically prevent.

Finally, while bugs in source are common, successful exploits are rare. Fuzzing campaigns often find a large number of software bugs, but rarely convert these bugs into meaningful exploits. Research teams are unlikely to disclose a simple out-of-bounds read in ZK, as most such bugs do not lead to meaningful system compromise. Real bug bounties and vulnerability research consists of demonstrating how to leverage a vulnerability into an exploit (*e.g.*, privilege escalation, arbitrary code execution, or reading protected memory). Proving these capabilities cannot be done with source alone and are intrinsically linked to the compiled binary and architecture. For example, Heath et al. [29] claim that they can prove the existence of vulnerabilities in `sed` and `gzip` despite using a Harvard architecture. While it is true that they can prove vulnerabilities on such an architecture, they would not be able to demonstrate that the vulnerability is exploitable if the exploit involved executing malicious code off the stack, since the machine would not be able to fetch instructions stored in RAM.

# 4 MODELING REAL-WORLD PROCESSORS

In this section we discuss the technical details of modeling our target real-world processor, MSP430. First we discuss the necessary modeling to cover the basic MSP430 processor semantics and then discuss additions to the processor semantics that are helpful when modeling exploits.

## 4.1 Modeling MSP430 Processor Semantics

The MSP430 is a ubiquitous microcontroller [42], making it the perfect target for proofs of exploit. The MPS430 architecture contains 27 instructions, 13 addressing modes, and 16 registers with 16-bit words. We design a circuit which models the state transition associated with each of these instructions. We note, however, that MSP430 is not a load/store architecture—unlike the processor designed for ZK proofs—which increases the complexity of modeling memory.

**Modeling Memory.** Prior work on ZK processors use load/store architectures to cleanly separate memory accesses and logical operations. This allows the RAM reduction to treat non-memory operations as no-ops when performing the memory consistency check and permutation proof. However, many real-world processors, such as the MSP430, use a variety of addressing modes that prevent such a clean distinction from being made. For example, consider the instruction add `add @r5, 2(r6)`, which adds the contents of memory at the address `r5` to the contents of memory at address `r6+2` and stores the result at address `r6+2`. Not only does this instruction both access memory and use the processor's ALU, but it actually performs two reads and a write.

Our processor model handles such instructions by augmenting each instruction in the program trace to include three *memory hints*, which are used by the decoded instruction and verified with the memory checker. The hints are separated into two *read hints*, `src` and `dst`, and a single *write hint*. The hints each contain the relevant information for the implicit load/store operations encoded into some instructions (*e.g.* the address and value of memory to read-/write). Specifically, the memory hints have the following structure:

- 1-bit On/Off indicator
- 16-bit Memory Address
- 19-bit Timestamp
- 1-bit Read/write indicator
- 1-bit Byte Mode indicator
- 1-bit Byte Mode Offset
- 16-bit Value

MSP430 supports byte operations on memory, so each memory hint indicates if it is in byte mode and the index of the byte on which the instruction is operating, if applicable.

Because MSP430 is a Von-Neumann architecture, fetching instructions constitutes a memory read. Each MSP430 instruction consists of a one-word opcode and up to two immediates, each of which requires its own read hint. Thus, the memory trace will contain six entries for each entry in the program trace. Checking these memory operations for consistency is straightforward, requiring only 194 AND gates per entry, so the memory checker requires 1,164 AND gates/cycle.

## 4.2 Interacting with the Program

In order to facilitate proofs of exploit, we choose to extend the base MSP430 ISA with cleanly modeled methods that allows the prover to interact with the program. Specifically, we are concerned with loading the program into the runtime, getting user inputs, and providing the program with entropy. While there are many potential ways to add these capabilities to the base ISA, we choose to add *system calls* that support these capabilities. This choice is inspired by the Microcorruption CTF challenges, which modeled system calls similarly in their version of the MSP430 ISA; by mirroring the choices made by the designers of the Microcorruption CTF challenges, we are able to "natively" support solutions for the challenges by directly mapping their syscalls onto our syscalls.

**Modeling System Calls.** System calls are an integral component of real-world software, providing the program access to key resources, including randomness, memory management, and user input. Many successful exploit strategies—and the techniques used to prevent such exploits—depend on the low-level details of syscall operations. For example, many processors implement memory protections such as ASLR by using system entropy to randomize the address space layout. Prior work on ZK processors ignores syscalls and does not provide the processor with randomness.

We provide a general approach to handling syscalls initiated via software interrupts. Our approach does not rely on adding new instructions or storing information in registers or memory, as this would change the low-level processor behavior we aim to preserve. Instead, we augment each trace entry with a 48-bit value that encodes a finite state machine representing the current syscall status. This finite state machine is fed to a co-processor which is only triggered once a software interrupt is called. When a syscall is triggered, the following sequence of events occurs:

(1) The processor freezes the register file, turns on the syscall flag, and loads the arguments and opcode into the syscall register.
(2) Execution continues, but the processor operates on the syscall register instead of the register file.
(3) Once the exit condition has been met, the syscall flag is turned off and normal execution resumes.

To better demonstrate this approach, we give the full details for our implementation of the `LOAD`, `GETS`, and `RAND` syscalls.

**Getting User Input.** Before program execution begins, the prover uses the `LOAD` syscall to pre-load their input into a special memory bank that is read-only once program execution begins. Pre-loading input is important for reasoning about exploits that circumvent ASLR and stack canaries, since knowing or influencing the random values used in such mitigations would make significant parts of the exploit trivial.

When the processor starts execution, the PC is set to the first instruction in the input binary, but the syscall co-processor is turned on and set to `LOAD`. The first instruction of the trace declares how many bytes of input will be loaded, and this value is placed in the syscall register. The processor will then continue to execute `LOAD` instructions, each time decrementing the syscall register until it reaches zero. At this point the syscall flag is turned off and program execution begins. At each step of the program, the processor checks that the prover cannot call `LOAD` after execution begins.

Once the input is pre-loaded, the processor accesses it via the GETS syscall. GETS takes two arguments off the stack: the address to which the input will be written, and the maximum allowed length of the input in bytes. The syscall will exit once a null byte is encountered or the maximum number of bytes is written.

When our MSP430 model encounters a call to GETS, the register file is frozen by turning on the syscall flag, and the target address and length are loaded into the syscall register. Subsequent clock cycles will use the memory hints in the trace to load user input byte-by-byte into memory, incrementing the address and decrementing the length variable in the syscall register. At each step, the input is checked for a null byte and the length variable is verified not to be zero. If either is zero, the syscall flag is turned off and normal processor execution resumes. Using this approach, the processor can emulate syscall operations — including the unrolling of variable length loops within the syscall logic — without altering the binary or memory state.

**Processor Entropy.** Our target version of MSP430 uses the RAND syscall to generate random values. In general, generation of high-entropy random values can be done using Fiat-Shamir. However, sometimes applications may use low-entropy random values, which cannot be generated using Fiat-Shamir while providing strong soundness guarantees, as the prover could grind to ensure that the randomness has the desired value. For example, 16-bit random values are used when calculating ASLR offsets and stack canaries. This limitation is inherent in the architecture itself — defenses that rely on low-entropy randomness will always be vulnerable to computationally powerful adversaries.

To provide some meaningful soundness in the case where low entropy defenses are used, we design our processor to naturally extend to *interactive* proofs in which the verifier can supply randomness directly. First, the prover commits to all inputs that will be fed into the program by loading these values into a special memory bank prior to program execution, as specified in the previous section. Then, the verifier supplies a random seed value seed from which all randomness for the RAND syscall will be generated.

Specifically, the processor executes a special GETRANDSEED syscall to load the verifier supplied randomness seed into an auxiliary RAND register. The GETRANDSEED syscall can only be called once and only after the initial LOAD syscall has finished executing. The processor circuit will fail if the prover attempts to call GETRANDSEED again.

Once the prover has completed the LOAD phase, they execute the following steps in the clear:

(1) Show the verifier that the PC is set to the program entry point, the syscall flag is turned on, and the syscall opcode is set to GETRANDSEED
(2) Acquire the randomness seed from the verifier
(3) Load the randomness into a public auxiliary RAND register
(4) Turn the syscall flag off

Since the syscall flag is turned off once GETRANDSEED is finished, program execution must proceed normally from the binary entry point. During each processor cycle, the prover will evaluate PRF(seed, step), where PRF is a pseudorandom function, and step is a counter indicating the number of processor cycles that have been executed. The first 16 bits of the output are then fed into the processor as the potential output of the RAND syscall. We emphasize that returning only 16 bits of randomness is inherent to the architecture. By making the prover commit to all their inputs to the program before learning the seed, they must commit to an exploit strategy that can work for any value of randomness generated. We repeat that the meaning of a proof of exploit that circumvents low-entropy protections is nuanced; we refer the reader back to Section 1.1 for a discussion.

Users are provided with the option to disable processor randomness, since many applications do not need this feature. Additionally, note that running these proofs interactively is only necessary when there are low-entropy defense mechanisms that the prover must overcome, like ASLR.

## 5 FORMALIZING EXPLOITS

Our aim is to provide vulnerability researchers with the necessary tools to precisely demonstrate exploits in real software without revealing underlying techniques. Therefore, we focus on creating a system that allows the prover to show that it knows some inputs such that running a public binary on those inputs on a real machine would result in a concrete exploit. This proof requires two components: demonstrating a given trace is valid, and demonstrating the trace triggered an exploit. The first component is handled using the previously discussed RAM reduction. We now discuss how exploits are shown during execution.

Many exploits can be detected by determining whether the attacker has arbitrary PC control. In this setting, the verifier challenges the prover to demonstrate they were able to produce a valid program trace concluding with the PC set to the challenge address. A similar protocol is used in the context of exploits that gain the ability to arbitrarily read or write memory.

A variety of exploits conclude with the execution of a syscall that should not have been accessible to the attacker. In an embedded systems context, this may manifest itself as turning on a microphone, turning off a security camera, or unlocking a door. This particular notion of exploit is relatively straightforward to formalize in a ZK context. The prover simply needs to demonstrate that at some point during a valid program execution, a known malicious syscall was executed. This can be checked at the processor level by checking at each step whether the syscall flag is on and then examining the syscall opcode as specified in Section 4. All of these checks can be fed to a large OR statement at the conclusion of the proof to demonstrate whether a malicious syscall was executed. As we discuss in Section 8, this is how we formalize the Microcorruption exploits, all of which conclude in a call to the special UNLOCK interrupt.

Proving privilege escalation exploits — exploits which allow the prover to execute commands with root privileges on the machine — is more complicated. Generally, this would involve calling the GETEUID syscall and demonstrating the output is 0, using a similar approach as above. However, this would require modeling a runtime environment complex enough to have a notion of user privileges. We leave modeling a complex runtime environment as important future work.

Generally speaking, our approach facilitates proofs about exploits that can be represented as a Boolean expression on each processor state across the entire program execution. All of the

above techniques are examples of this broader paradigm (*e.g.*, there exists a step of execution such that the instruction loaded by the processor is a malicious syscall). While this approach is sufficiently general to cover most common exploits, it has some fundamental limitations. In particular, our proof of exploit toolchain is incapable of reasoning about exploits that rely on microarchitectural bugs such as Spectre and Meltdown. Similarly, a Row Hammer type attack would also be out of scope since unintended physical properties of RAM cannot be simulated within a ZK context. Fortunately, most real-world exploits do not rely on microarchitectural bugs, so we do not view this as a major limitation.

**Barriers to Easy Use.** Although our toolchain allows provers to produce proofs for any predicate over the processor states, the process of selecting the *right* predicate may be non-trivial—especially for vulnerability researchers without zero-knowledge expertise. Indeed, in our envisioned workflow (Section 1), we imagine that a sponsor might post a bug bounty to which vulnerability researchers could respond. One approach would be to have the bug bounty itself formalize the statement to prove in zero-knowledge; this approach is implicitly used in the Microcorruption CTF exercises, as the `UNLOCK` syscall is part of the challenge description. In more complex systems, there may be a huge number of potential processor states that would be considered problematic, such that enumerating all the processor states would be impractical. In such cases, the burden of selecting the correct statement—and demonstrating the statement's importance—would fall to the vulnerability researcher. Making these processes easier is important future work.

## 6 CIRCUIT COMPILER

The ZK proof system that we target accepts statements as either Boolean or arithmetic circuits. There are several tools created specifically for ZK statement generation such as Frigate [40], libsnark [34], and Circom [4], but they mostly target arithmetic circuits, which are not performant when handling real-world processor models. Frigate synthesizes code written in a subset of C. However, we found that it did not give us the granularity necessary to optimize circuits for MSP430.

Instead, we chose to write our processor circuit in Verilog, a widely used hardware description language (HDL) with mature open-source tooling. In particular, we used Yosys [53] to synthesize our core circuit components and Icarus Verilog for simulation and testing. Using Verilog allowed us to divide our RAM reduction into a collection of discrete Boolean modules, including the single-step MSP430 processor circuit and the memory consistency checker. We use Yosys to synthesize these components to a BLIF [19] file that encodes the hierarchical arrangement of the components and their logic gates. Finally, we assemble these components into a flat, non-hierarchical encoding of the RAM reduction in the Bristol Fashion [2] using a circuit flattening library.

We designed our in-house flattener to take advantage of the fact that our circuit is highly structured, so we can aggressively cache flattened versions of the components and avoid repeating work. Using this approach of flattening components once and stapling them together, our flattening library can assemble the full RAM reduction for traces with 7k steps in 6 minutes using 20GB of RAM—an improvement of 99% in running time and 88% in RAM usage over using Yosys for flattening.

As described in Section 7.1, our permutation proof is prohibitively complex to be evaluated via a Boolean circuit, so we elected to specify it via an arithmetic circuit on $\mathbb{Z}_{2^{64}}$. Yosys and Verilog are only designed to operate on Boolean circuits, which presents a problem because using a HDL like Verilog is substantially easier than working at the level of individual gates when designing complex circuits.

We, however, use blackbox modules—a feature of Yosys designed to connect circuits to unknown hardware—to create models for the arithmetic logic gates in Verilog, which we then used to specify our permutation proof circuit. While we still had to ultimately specify the circuit at the gate level, working in Verilog broke up the circuit into hierarchical modules and assigned names to wires, greatly reducing debugging time.

After synthesizing the permutation circuit to a BLIF file, we pass it to our circuit compositor—a modified version of the circuit flattener that can accept a flattened Boolean circuit and a flattened arithmetic circuit and generate a specification for connecting the outputs of the Boolean circuit to the inputs of the arithmetic circuit using specialized BooleanToArithmetic gates. The 3-tuple of circuits consisting of the Boolean circuit, the connection circuit, and the arithmetic circuit is then passed to Reverie, which evaluates it as the complete ZK statement.

## 7 CRYPTOGRAPHIC OPTIMIZATIONS

**Choice of Proof System.** To instantiate our toolchain and optimize our proof system, we must first select a proof system. A number of considerations are relevant when selecting a suitable proof system for our particular application, most notably: (1) Prover/Verifier Complexity: Many widely deployed ZK proof systems are based on succinct non-interactive arguments of knowledge (SNARKs) (*e.g.* [25, 44]), which produce compact proof size at the expense of high prover runtime, complicated knowledge assumptions, and a trusted setup phase. While these tradeoffs are practical for space-limited applications, *e.g.* decentralized ledgers, the overhead of this approach would limit the complexity of RAM programs and exploits about which we could reason. Therefore we prioritize reducing concrete *prover time* rather than bandwidth. In order to somewhat offset the larger proof size we ensure that proofs can be verified in a streaming manner, meaning the verifier can process the proof as he is downloading it (without storing it). (2) Interactive vs Non-interactive: While interactive (private-coin) proofs systems can enable more efficient/flexible proofs, we opt for non-interactive proofs to enable a wider variety of use cases, as discussed in the introduction. This includes posting the proof for public verification and inclusion in long-term bug tracking logs. Non-interactivity can also be valuable when the prover may no longer be online or moving proofs across air-gaps (security researchers might be wary about allowing arbitrary people to open connections to the server holding the sensitive zero-day exploit).

These considerations lead us to believe that the KKW proof system [33] is well-suited for our application. We give a summary of MPC-in-the-head and the KKW proof system in Appendix A.

**Input.** Secret lists $A = \{A_i\}_{i\in[\ell]}$ and $B = \{B_i\}_{i\in[\ell]}$.
**Public Input.** A random challenge $x \in \mathbb{Z}_q$.
**Circuit.** Compute and compare

$$\prod_{i=1}^{\ell}(A_i - x) \stackrel{?}{=} \prod_{i=1}^{\ell}(B_i - x)$$

**Output.** 1 if the above check is valid, 0 otherwise.

**Figure 2: Unknown Permutation Proof Circuit ($C_{\text{shuffle}}$). The circuit checks if two secret lists are permutations of each other.**

Throughout this section we denote the party that executes the preprocessing in KKW as $P_0$ and use $n$ to denote the number of parties in the MPC. We note that several improvements to the initial KKW system have been proposed recently, *e.g.* [5, 18], that could be integrated into our approach in future work.

## 7.1 Memory Permutation Proof (over $\mathbb{Z}_q$)

An unknown permutation proof is a zero-knowledge proof of knowledge that shows that the prover has two lists that are a permutation of each other, *i.e.* list $A = \{A_i\}_{i\in[\ell]}$ and $B = \{B_i\}_{i\in[\ell]}$ such that $\pi(A) = B$ for some permutation $\pi$. As the verifier does not know the lists nor the permutation, the proof is done with respect to a commitment to each list. We require an unknown permutation proof that will be efficient within MPC-in-the-head.

We implement the unknown permutation proof using the circuit defined in Figure 2 over a large ring, based on techniques first introduced by Bootle et al. [15], and first explored by Neff [41]. This stand-alone circuit receives two secret shared lists and a public randomly selected challenge $x$. Within the circuit, we view $A$ and $B$ as the set of roots of two polynomials, evaluate them at $x$ and check equality, *i.e.* asserting $\prod_i(A_i - x) = \prod_i(B_i - x)$. Intuitively, perfect completeness follows on the commutativity of multiplication, while statistical soundness relies on the Swartz-Zippel lemma stating that two polynomials with distinct roots share an evaluation at a random point only with small probability[5].

For soundness, the random challenge $x$ must be selected after the prover has committed to the secret shared lists, however the subsequent computation depends on the challenge. We accommodate this by introducing an additional round (5 rounds total)[6] in which the verifier samples $x$, after the prover has committed to the inputs/witness, but before committing to the views of every party.

Theorem 1 (Unknown Permutation Proof). *Given two lists $A$ and $B$ with $\ell$ elements in $\mathbb{Z}_q$ and an instance of the KKW protocol with $n$ participants and $m$ preprocessing repetitions. Using the above circuit and the challenge input inside a KKW protocol is an honest-verifier ZKPoK to prove knowledge of two lists $A$ and $B$ such that there exists a permutation $\pi$ such that $\pi(A) = B$ with soundness/knowledge error* $\max\left\{\frac{1}{m}, \frac{1}{n} + \frac{\ell}{q-1} - \frac{\ell}{q-1}\frac{1}{n}\right\}$.

The proof of this theorem can be found in Appendix B.1.

[5]When the size of the field dominates the degree of the polynomials. Note we do not need the soundness error to be negligible, but only to be dominated by $^{n-1}/_{n}$ from KKW.

[6]We reason that this additional round does not affect the knowledge error of the Fiat-Shamir transform, compared to the original 3 rounds. Note that, in general, soundness of the Fiat-Shamir transform decreases exponentially in the number of rounds.

When amplifying the soundness by parallel repetitions, the soundness error of the permutation proof is dominated by the soundness error of KKW. As such, it is straightforward to observe that using this permutation proof does not introduce the need for any additional repetitions of the proof. We show this formally in Appendix C, along with discussing the technical detail of performing these operations in a ring, rather than a field

## 7.2 Ring Switching

One drawback of the permutation proof described in the previous section is that it relies on a large field/ring for soundness which leads to inefficient proofs of Boolean circuits. Unfortunately, real-world processors are most efficiently realized as Boolean circuits that pay a high cost for multiplication gates. The permutation proof can be implemented in a Boolean circuit by simulating a larger ring, however the $\log^2(q)$ overhead introduced by simulating the ring multiplication negates the improvements over the routing network used in the work of Ben-Sasson et al. To avoid simulating arithmetic in a large ring, while still enabling application logic (CPU specification) to be proved using a Boolean circuit we rely on ring-switching techniques: enabling us to switch/pack a collection of Booleans into an element in a ring of sufficiently large order. This technique introduces an overhead of 3 AND-gates for every bit that needs translating. In our case, where we will switch to $\mathbb{Z}_{2^{64}}$, this means 192 AND-gates for every element in both lists. We base our ring-switching technique on the use of *edaBits* as introduced by Escudero et al. [20], which in turn was based on *daBits* by Rotaru and Wood [47]. We will apply the preprocessing optimization of KKW to achieve these results.

**Preprocessing.** Let $\xi$ be the number of bits required to represent values of the larger field. During the preprocessing phase, we generate secret shares for the MPC players of the correlated random values $r$ and $r_0, \ldots, r_{\xi-1}$, where $r$ is a value in the larger ring and $r_0, \ldots, r_{\xi-1}$ are Boolean values, subject to the constraint $r = \sum_{i=0}^{\xi-1} r_i 2^i$, in the larger field. Thus, the players receive Boolean sharings $[r_0], \ldots, [r_{\xi-1}]$ and an arithmetic sharing $[\![r]\!]$. Note that none of the participants have any of the values $r, r_0, \ldots, r_{\xi-1}$ in the clear, they only possess a share of these values. Generation of this correlated randomness can be done using the same techniques used for Beaver triple generation in KKW: the dealer ($P_0$) generates and "sends" the shares to the respective players.

**Online.** The translation of $([x_0], \ldots, [x_\xi])$ into $[\![x]\!]$ with

$$x = \sum_{i=0}^{\xi} x_i 2^i$$

is done in the following way:

(1) In the Boolean circuit compute the $\mathbb{Z}_q$ addition of $r + x$ using a full adder, *i.e.* compute:

$$[(x+r)_0], \ldots, [(x+r)_{\xi-1}] = ([x_0], \ldots, [x_{\xi-1}]) +_{\mathbb{Z}_q} ([r_0], \ldots, [r_\xi]))$$

**Table 1: Benchmarks for proofs of exploits (at 128 bits of security) for a representative subset of the Microcorruption exercises. The selected exercises cover the most important exploit categories, including buffer overflow, code injection, and bypassing memory protection. These exercises are ordered by the difficulty of the exercise, as estimated by the Microcorruption creators.**

| Exercise Name | Processor Cycles | Prover (sec) | Verifier (sec) | Size (mb) | Exploit Type |
|---|---|---|---|---|---|
| New Orleans | 2392 | 22 | 7 | 295 | Password embedded in binary |
| Hanoi | 6199 | 25 | 18 | 322 | Buffer overflow |
| Cusco | 5178 | 21 | 15 | 269 | Buffer overflow |
| Montevideo | 6676 | 28 | 20 | 358 | Code injection via `strcpy` bug |
| Johannesburg | 6311 | 26 | 19 | 332 | Stack cookie bypass |
| Santa Cruz | 12835 | 754 | 39 | 680 | Code injection via `strcpy` bug |
| Addis Ababa | 5360 | 23 | 17 | 296 | Format string vulnerability |
| Novosibirsk | 19833 | 89 | 63 | 1100 | Format string vulnerability |
| Vladivostok | 50823 | 454 | 152 | 6048 | ASLR bypass |

(2) Reconstruct the masked bits $(x+r)_0, \ldots (x+r)_{\xi-1} \in \mathbb{Z}_2$, lift the bits to the ring $\mathbb{Z}_q$ and and convert the decomposition into $x + r \in \mathbb{Z}_q$ by publically computing the linear combination: $x' = x + r = \sum_{i=0}^{\xi-1} 2^i (x+r)_i \in \mathbb{Z}_q$
(3) In the arithmetic circuit subtract the randomness $r$ from $x'$ the input coming from the Boolean circuit, *i.e.* $x = x' - r$.

Note that only (1) has non-linear (over $\mathbb{Z}_2$) operations.

Theorem 2 (Ring Switching). *Given a Boolean circuit $C_{\mathbf{bool}}$ and an arithmetic circuit $C_{\mathbf{arith}}$ that need to be run consecutively, a definition of which output wires from $C_{\mathbf{bool}}$ are going into $C_{\mathbf{arith}}$, and an instance of the KKW protocol with n participants and m preprocessing repetitions. The above protocol is an honest-verifier ZKPoK with soundness/knowledge error* $\max\left\{\frac{1}{m}, \frac{1}{n}\right\}$.

The proof of this theorem can be found in Appendix B.2. Note that the soundness error is exactly the same as for the original KKW protocol, therefore, no extra iterations of the protocol are needed because of the addition of the ring switching.

## 8 IMPLEMENTATION AND EVALUATION

**Reverie.** Our prover 'Reverie' [43] is an optimized implementation of the KKW [33] proof system in the Rust programming language. Reverie is generic and can be instantiated over any commutative ring. Reverie optimizes KKW for our particular application as follows:

- **Streaming.** Rather than compute the correlated randomness for the entire circuit before evaluation, Reverie interleaves the preprocessing with the online execution: in effect player $P_0$ is implemented as a coroutine. This avoids storing all the preprocessed material in memory.
- **Bit Slicing.** Every online player in KKW executes the same simple operation during the evaluation of addition and multiplication gates, hence bit-slicing 'across the players' allows executing every player in parallel, *e.g.* for the ring $\mathcal{R} = \mathbb{F}_2$ and $n = 64$ the values of two wires can be added using a single XOR of 64-bit integers.
- **Shadowing.** The model of execution in 'Reverie' is a straight-line RAM program: there is an array of cells and a program consists of a list of `Input`/`Add`/`Mul`/`Output` instructions reading/writing to cells. A circuit is a straight-line program in single assignment form (*i.e.* every cell is only written to once). Since the execution of a CPU is very local, this allows us to reclaim memory by overwriting cells, in practice reclaiming > 95% over naïvely loading the circuit.
- **Parallel.** KKW requires many repetitions for soundness, these are executed in parallel.

**Table 2: Comparative Measurements for NIZKs computing 511 iterations of SHA256 (Merkle tree with 256 leaves). Measurements for prior work from [54] on an Amazon EC2 c5.9xlarge with 70GB of RAM and Intel Xeon platinum 8124m CPU with 18 3GHz virtual cores. Because these proof systems and implementations were unable to exploit parallelism, all benchmarks were run on a single thread. Reverie was benchmarked on a Digital Ocean virtual machine with 32 virtual cores and 256GB of memory. We note that our choice of protocol and our implementation is able to take advantage of the parallelism offer by the multiple cores, which is part of the reason Reverie is able to dramatically out-perform prior work.**

| Proof System | Gen (sec) | Prove (sec) | Ver (sec) | Size (KB) |
|---|---|---|---|---|
| Aurora [12] | - | 3,199 | 15.2 | 174.3 |
| Bulletproofs [16] | - | 2,555 | 98 | 2 |
| libSTARK [7] | - | 2,022 | 0.044 s | 395 |
| Hyrax [50] | - | 1,041 | 9.9 | 185 |
| Ligero [1] | - | 400 | 4 | 1,500 |
| libSNARK [13] | 1027 | 360 | 0.002 | .013 |
| Libra [54] | 210 | 201 | 0.71 | 51 |
| Reverie (This Work) | - | 8 | 7.67 | 113,848 |

All of these optimizations contribute to Reverie's exceptionally fast performance. Reverie is able to prove 511 iterations of SHA256 in 8 seconds. We compare this to the benchmarks reported in prior work from [54] in Table 2. We note that these are not strictly apples-to-apples comparisons as we were unable to control for the benchmarking environment for prior work. However, we note that Reverie does strikingly well. Libsnark requires 1,387 seconds, Bulletproofs requires 2,555 seconds, and Ligero requires 400 seconds (see Table 2). The proofs generated by Reverie are larger than the other three, but since it supports streaming all that is required is a network connection between prover and verifier with modest bandwidth.

**Table 3: Breakdown of processor circuit components**

| Component | Non-linear Gates Per Instruction |
|---|---|
| Memory checker | 1,164 |
| Permutation proof | 2,280 |
| Processor | 7,247 |
| Decoder | 568 |
| ALU | 549 |
| Hint verifier | 237 |
| Operand fetching | 2,176 |
| Register file | 2,880 |

**Proofs of Exploitability: Microcorruption.** We chose to use the Microcorruption CTF as a benchmark set for our ZK proof of vulnerability system. The CTF challenges involve hacking a smart lock controlled by an MSP430 using common exploitation techniques such as buffer overflows, code injection, and bypassing memory protections. While the challenges contain a wide variety of bugs, ultimately they all conclude with a call to the `UNLOCK` system call. For example, the Addis Ababa challenge can be solved by using a format string vulnerability to overwrite a segment of memory that contains information about whether the correct password was entered or not, leading to a successful call to the `UNLOCK` system call.[7].

Therefore our ZK proofs of vulnerability check both that the witness trace is valid, and that at least one step of execution was a call to the `UNLOCK` system call. An advantage of this approach is that all ZK performance metrics are linear in the trace size, regardless of exploit technique.

**Performance.** We present benchmarks for a representative set of the Microcorruption exercises in Table 1. This set of benchmarks covers many of the most important exploit types, including buffer overflow, code injection, and bypassing memory protection. Each of these benchmarks was computed on a Digital Ocean virtual machine with 32 virtual cores and 256GB of memory. We found that our implementation produces a proof for 216 MSP430 instructions every second. Overall, each instruction requires 10,691 AND gates to execute. In Table 3, we give a breakdown of the gate count for each component of the RAM reduction, along with the major components of the processor. Although the resulting proofs produced are large and may take a non-trivial time to create, we note that these resources and time are insignificant compared to the effort it takes to develop the exploit and the time that the parties would spend negotiating disclosure.

## 9 CONCLUSION

We have presented a toolchain that can practically prove knowledge of real exploits for real-world processor architectures without the need for source code. Our approach offers a concrete solution to a real-world problem: how should vulnerability researchers demonstrate their capabilities to the managers of bug bounty programs? Using our proof system, the managers of bug bounty programs need not be concerned that vulnerability researchers are overstating their findings and vulnerability researchers are protected against preemptive disclosure. Moreover, our techniques can be used to enhance the current bug bounty ecosystem by allowing robust, trustworthy public disclosure of vulnerabilities without handing attackers live exploits. Given the importance of bug bounty programs to security critical software, we believe that our work represents a significant step forward.

[7]For more details about the Microcorruption challenges, we point the reader to https://microcorruption.com or the reference manual [36]

## ACKNOWLEDGMENTS

This work is supported by DARPA under agreement No. HR0011-20C0084. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA. The first author is also supported in part by NSF under awards CNS-1653110, and CNS-1801479, the Office of Naval Research under contract N00014-19-1-2292, as well as a Security and Privacy research award from Google. The second author is also funded by Concordium Blockchain Research Center, Aarhus University, Denmark. The forth author is also supported by the National Science Foundation under Grant #2030859 to the Computing Research Association for the CIFellows Project and is also supported by DARPA under Agreement No. HR00112020021.

## REFERENCES

[1] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM CCS 2017: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 2087–2104. https://doi.org/10.1145/3133956.3134104

[2] David Archer, Victor Arribas Abril, Steve Lu, Pieter Maene, Nele Mertens, Danilo Sijacic, and Nigel Smart. 2022. 'Bristol Fashion' MPC Circuits. https://homes.esat.kuleuven.be/~nsmart/MPC/.

[3] V. Arvind, P. Mukhopadhyay, and S. Srinivasan. 2008. New Results on Noncommutative and Commutative Polynomial Identity Testing. , 268-279 pages. https://doi.org/10.1109/CCC.2008.22

[4] 0Kims Association. 2018. Circom: a circuit compiler for zkSNARKs. https://github.com/iden3/circom.

[5] Carsten Baum and Ariel Nof. 2020. Concretely-Efficient Zero-Knowledge Arguments for Arithmetic Circuits and Their Application to Lattice-Based Cryptography. In *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part I (Lecture Notes in Computer Science, Vol. 12110)*, Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas (Eds.). Springer, Heidelberg, Germany, Edinburgh, UK, 495–526. https://doi.org/10.1007/978-3-030-45374-9_17

[6] Assaf Ben-David, Noam Nisan, and Benny Pinkas. 2008. FairplayMP: a system for secure multi-party computation. In *ACM CCS 2008: 15th Conference on Computer and Communications Security*, Peng Ning, Paul F. Syverson, and Somesh Jha (Eds.). ACM Press, Alexandria, Virginia, USA, 257–266. https://doi.org/10.1145/1455770.1455804

[7] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046. https://eprint.iacr.org/2018/046.

[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable Zero Knowledge with No Trusted Setup. In *Advances in Cryptology – CRYPTO 2019, Part III (Lecture Notes in Computer Science, Vol. 11694)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 701–732. https://doi.org/10.1007/978-3-030-26954-8_23

[9] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 459–474. https://doi.org/10.1109/SP.2014.36

[10] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. 2013. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In *ITCS 2013: 4th Innovations in Theoretical Computer Science*, Robert D. Kleinberg (Ed.). Association for Computing Machinery, Berkeley, CA, USA, 401–414. https://doi.org/10.1145/2422436.2422481

[11] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Advances in Cryptology – CRYPTO 2013, Part II (Lecture Notes in Computer Science, Vol. 8043)*, Ran Canetti and Juan A. Garay (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 90–108. https://doi.org/10.1007/978-3-642-40084-1_6

[12] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. 2019. Aurora: Transparent Succinct Arguments for R1CS. In *Advances in Cryptology – EUROCRYPT 2019, Part I (Lecture Notes in Computer Science, Vol. 11476)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, Germany, Darmstadt, Germany, 103–128. https://doi.org/10.1007/978-3-030-17653-2_4

[13] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security 2014: 23rd USENIX Security Symposium*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, San Diego, CA, USA, 781–796.

[14] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. 2020. Ligero++: A New Optimized Sublinear IOP. In *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 2025–2038. https://doi.org/10.1145/3372297.3417893

[15] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. 2018. Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution. In *Advances in Cryptology – ASIACRYPT 2018, Part I (Lecture Notes in Computer Science, Vol. 11272)*, Thomas Peyrin and Steven Galbraith (Eds.). Springer, Heidelberg, Germany, Brisbane, Queensland, Australia, 595–626. https://doi.org/10.1007/978-3-030-03326-2_20

[16] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 315–334. https://doi.org/10.1109/SP.2018.00020

[17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. 2017. Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives. In *ACM CCS 2017: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 1825–1842. https://doi.org/10.1145/3133956.3133997

[18] Cyprien de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. 2021. Limbo: Efficient Zero-knowledge MPCitH-based Arguments. In *ACM CCS 2021: 28th Conference on Computer and Communications Security*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 3022–3036. https://doi.org/10.1145/3460120.3484595

[19] Nikos Drakos and Ross Moore. 1992. Berkeley Logic Interchange Format (BLIF).

[20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *Advances in Cryptology – CRYPTO 2020, Part II (Lecture Notes in Computer Science, Vol. 12171)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 823–852. https://doi.org/10.1007/978-3-030-56880-1_29

[21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. 2021. Constant-Overhead Zero-Knowledge for RAM Programs. Cryptology ePrint Archive, Report 2021/979. https://eprint.iacr.org/2021/979.

[22] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. 2016. ZKBoo: Faster Zero-Knowledge for Boolean Circuits. In *USENIX Security 2016: 25th USENIX Security Symposium*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, Austin, TX, USA, 1069–1083.

[23] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1986. Proofs that Yield Nothing But their Validity and a Methodology of Cryptographic Protocol Design (Extended Abstract). In *27th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Toronto, Ontario, Canada, 174–187. https://doi.org/10.1109/SFCS.1986.47

[24] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Prove all NP-Statements in Zero-Knowledge, and a Methodology of Cryptographic Protocol Design. In *Advances in Cryptology – CRYPTO'86 (Lecture Notes in Computer Science, Vol. 263)*, Andrew M. Odlyzko (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 171–185. https://doi.org/10.1007/3-540-47721-7_11

[25] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016, Part II (Lecture Notes in Computer Science, Vol. 9666)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, Heidelberg, Germany, Vienna, Austria, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11

[26] NCC Group. 2013. Microcorruption: Embedded Security CTF. https://microcorruption.com.

[27] David Heath and Vladimir Kolesnikov. 2020. A 2.1 KHz Zero-Knowledge Processor with BubbleRAM. In *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 2055–2074. https://doi.org/10.1145/3372297.3417283

[28] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. In *Advances in Cryptology – EUROCRYPT 2020, Part III (Lecture Notes in Computer Science, Vol. 12107)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, Germany, Zagreb, Croatia, 569–598. https://doi.org/10.1007/978-3-030-45727-3_19

[29] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. 2021. Zero Knowledge for Everything and Everyone: Fast ZK Processor with Cached ORAM for ANSI C Programs. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1538–1556. https://doi.org/10.1109/SP40001.2021.00089

[30] Texas Instruments. 2006. MSP430x1xx Family User Guide. https://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[31] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2007. Zero-knowledge from secure multiparty computation. In *39th Annual ACM Symposium on Theory of Computing*, David S. Johnson and Uriel Feige (Eds.). ACM Press, San Diego, CA, USA, 21–30. https://doi.org/10.1145/1250790.1250794

[32] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM CCS 2013: 20th Conference on Computer and Communications Security*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 955–966. https://doi.org/10.1145/2508859.2516662

[33] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. 2018. Improved Non-Interactive Zero Knowledge with Applications to Post-Quantum Signatures. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 525–537. https://doi.org/10.1145/3243734.3243805

[34] SCIPR Lab. 2012-2020. libsnark: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark.

[35] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - Secure Two-Party Computation System. In *USENIX Security 2004: 13th USENIX Security Symposium*, Matt Blaze (Ed.). USENIX Association, San Diego, CA, USA, 287–302.

[36] Microcorruption. 2013. Lockitall LockIT Pro User Guide. https://microcorruption.com/public/manual.pdf.

[37] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. 2013. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 397–411. https://doi.org/10.1109/SP.2013.34

[38] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[39] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. 2016. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. , 112–127 pages.

[40] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. 2016. Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation. , 112-127 pages. https://doi.org/10.1109/EuroSP.2016.20

[41] C. Andrew Neff. 2001. A Verifiable Secret Shuffle and Its Application to e-Voting. In *ACM CCS 2001: 8th Conference on Computer and Communications Security*, Michael K. Reiter and Pierangela Samarati (Eds.). ACM Press, Philadelphia, PA, USA, 116–125. https://doi.org/10.1145/501983.502000

[42] Emmanuel Odunlade. 2020. Top 10 popular microcontrollers among makers. https://www.electronics-lab.com/top-10-popular-microcontrollers-among-makers/.

[43] Trail of Bits. 2022. Reverie: An efficient and generalized implementation of the IKOS-style KKW proof system. https://github.com/trailofbits/reverie.

[44] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 238–252. https://doi.org/10.1109/SP.2013.47

[45] Rafael Pass and abhi shelat. 2010. A Course In Cryptography. https://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf.

[46] Ryan Pickren. 2021. Hacking the Apple Webcam (again). https://www.ryanpickren.com/safari-uxss.

[47] Dragos Rotaru and Tim Wood. 2019. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In *Progress in Cryptology - INDOCRYPT 2019: 20th International Conference in Cryptology in India (Lecture Notes in Computer Science, Vol. 11898)*, Feng Hao, Sushmita Ruj, and Sourav Sen Gupta (Eds.). Springer, Heidelberg, Germany, Hyderabad, India, 227–249. https://doi.org/10.1007/978-3-030-35423-7_12

[48] Yannis Smaragdakis. 2019. Sound Analysis: Can We Tell the Truth About Programs? https://blog.sigplan.org/2019/09/18/sound-analysis-can-we-tell-the-truth-about-programs/.

[49] swisspost evoting. 2019. E-Voting System 2019. https://gitlab.com/swisspost-evoting/e-voting-system-2019.

[50] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. 2018. Doubly-Efficient zkSNARKs Without Trusted Setup. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 926–943. https://doi.org/10.1109/SP.2018.00060

[51] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. Available At https://github.com/emp-toolkit.

[52] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2020. Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits. Cryptology ePrint Archive, Report 2020/925. https://eprint.iacr.org/2020/925.

[53] Claire Xenia Wolf. 2012-2022. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.

[54] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *Advances in Cryptology – CRYPTO 2019, Part III (Lecture Notes in Computer Science, Vol. 11694)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 733–764. https://doi.org/10.1007/978-3-030-26954-8_24

[55] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In *ACM CCS 2021: 28th Conference on Computer and Communications Security*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 2986–3001. https://doi.org/10.1145/3460120.3484556

[56] Greg Zaverucha. 2020. The Picnic Signature Algorithm. https://github.com/microsoft/Picnic/raw/master/spec/spec-v3.0.pdf.

[57] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. vRAM: Faster Verifiable RAM with Program-Independent Preprocessing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 908–925. https://doi.org/10.1109/SP.2018.00013

# A KKW18 MPC-IN-THE-HEAD

Ishai et al. [31] demonstrated that it is possible to construct ZKPs from secure multiparty computation (MPC). Their technique, commonly called MPC-in-the-head or IKOS, has since inspired several concretely efficient concrete protocols, including ZKBoo [22], ZKB++ [17], KKW18 [33], and Ligero [1, 14].

In IKOS, the prover first secret shares the witness among $n$ virtual parties, and then emulates the MPC execution for the computation of the NP predicate on the (secret shared) witness among the virtual parties. The prover then commits to each emulated party's view, and the verifier then selects $n' \subset n$ views to check for consistency, where $n'$ is smaller than the MPC's privacy threshold. When the MPC is semi-honest the knowledge error is $n'/n$, and parallel repetition can be used to amplify soundness. IKOS is both flexible and can be made non-interactive using the Fiat-Shamir heuristic.

The KKW [33] proof system is an instantiation of the IKOS framework, using a semi-honest, dishonest-majority ($n' = n - 1$) MPC protocol in the broadcast setting using additive sharings over any commutative ring. In this MPC protocol, the (emulated) players compute an arithmetic circuit in a gate-by-gate manner: for each wire $\alpha$, the players $P_1, \ldots, P_n$ maintain additive shares $[m_\alpha]^{(1)}, ..., [m_\alpha]^{(n)}$ of a 'mask' $m_\alpha = \sum_i [m_\alpha]^{(i)}$. The value $z_\alpha$ assigned to the $\alpha$ wire is masked as $\hat{z}_\alpha = z_\alpha - m_\alpha$ and the 'correction' $z_\alpha$ is known to all players. Linear operations are executed locally by the players, while multiplication of wire values is handled using standard Beaver multiplication.

Because the prover's evaluation of the circuit is privacy-free, the MPC protocol generates the Beaver triples in a privacy-free way using a central coordinator. We denote this special player that generates the preprocessing as $P_0$. This coordinator only distributes preprocessing to the other players and does not participate in the online evaluation.

To ensure honest behavior by the coordinator KKW relies on cut-and-choose: the prover runs the MPC protocol many times, and in a subset of the executions the verifier opens and checks the view of $P_0$ (*i.e.* checks that the preprocessing has been done honestly), in the remaining executions the verifier opens an $n - 1$ size subset of the players $P_1, \ldots, P_n$ and checks these views for consistency.

# B PROOFS

## B.1 Proof of Theorem 1

Perfect completeness follows from the completeness of the KKW protocol as well as from the correctness of the circuit, which can be easily verified by inspection. Therefore, we will focus on proving honest-verifier zero-knowledge and soundness.

To prove that this protocol achieves perfect zero-knowledge, we can take the simulator $\mathcal{S}_{\mathbf{KKW}}$ that was used in KKW. The only change we have to make is that the simulator also chooses the challenge $x \in \mathbb{Z}_q$ uniformly at random. The same hybrid argument can be used as in the original proof. Given that the original simulator was indistinguishable from a real execution, we can conclude that this simulator is also indistinguishable from a real execution.

Similarly, to prove witness extraction, we can use the witness extractor from KKW. Note that after a full run of the protocol we have all messages as if we ran a normal KKW protocol for the circuit $C_{\mathbf{shuffle}}$, with a public input $x$, *i.e.* we don't have to extract $x$ because it is part of the transcript. Hence, we can use the witness extractor as described in KKW to extract $A$ and $B$, such that $\pi(A) = B$, for some permutation $\pi(\cdot)$.

The soundness error induced by the shuffle proof is $\frac{\ell}{q-1}$, which follow directly from the Schwartz-Zippel lemma. To see this, note that the $x$ is selected at random and the number of points that are shared by the two polynomials is bounded by their degree $\ell$. The soundness of the MPC-in-the-head protocol is $\max\left\{\frac{1}{m}, \frac{1}{n}\right\}$, as we are only considering the non-amplified version of KKW. To violate soundness, the prover must either succeed in the cheating during the preprocessing or the online phase. During the preprocessing, the probability is $\frac{1}{m}$. During the online phase, either the prover must cheat or produce an invalid shuffle proof. The probability of this happening is $\frac{1}{n} + \frac{\ell}{q-1} - \frac{\ell}{q-1}\frac{1}{n}$. Therefore, the overall soundness error is $\max\left\{\frac{1}{m}, \frac{1}{n} + \frac{\ell}{q-1} - \frac{\ell}{q-1}\frac{1}{n}\right\}$.

## B.2 Proof of Theorem 2

Completeness follows immediately from the completeness of the KKW protocol as well as the basic arithmetic used for transforming output from the boolean circuit to input to the arithmetic circuit.

To show perfect zero-knowledge we build the following simulator:

- Use the simulator $\mathcal{S}_{\mathbf{KKW}}$ on $C_{\mathbf{bool}}$,
- Actually do the transformation as it is done in the real protocol.
- Use the simulator $\mathcal{S}_{\mathbf{KKW}}$ on $C_{\mathbf{arith}}$,

Because $\mathcal{S}_{\mathbf{KKW}}$ generates a proof transcript that is indistinguishable form a real proof, and the second step is done exactly like it is done in the real protocol, we can conclude that this new simulator also produces a proof transcript that is indistinguishable from a real execution.

Witness extraction can be shown by first extracting the witness of the second circuit, and then using that witness for extracting the witness of the first circuit, which is also the witness for the complete circuit.

Soundness error is the maximum between both circuits of the soundness error as computed in KKW. To achieve better soundness, we can choose the number of executions according to the circuit with the worst soundness error.

# C KNOWLEDGE ERROR OF PERMUTATION PROOF

When amplifying the soundness by parallel repetitions, the soundness error of the permutation proof is dominated by the soundness error of KKW. We computed the soundness error for several different parameters, *i.e.* changing the size of the arithmetic group, the number of players, and the number of repetitions, these results are shown in Table 4 and Figure 3. Hence, we can optimize for speed and proof size while targeting an error of $\epsilon \leq 2^{-128}$. Similar to computing the number of repetitions required for the KKW protocol, we can compute the number of parallel repetitions required to amplify the soundness of the permutation proof. The probability of a cheating prover passing the preprocessing phase is:

$$\max_{m-\tau \leq k \leq m} \left\{ \binom{k}{m-\tau} \cdot \binom{m}{m-\tau}^{-1} \right\}$$

Which is exactly as shown in KKW [33]. Conditioned on the cheating prover passing the preprocessing phase, the probability of passing the online phase is:

$$\max_{m-\tau \leq k \leq m} \left\{ \left( \frac{1}{n} + \frac{\ell}{q-1} - \frac{\ell}{q-1}\frac{1}{n} \right)^{m-\tau-k} \right\}$$

With $\ell$ the number of elements in the lists, and $q$ the order of the field in which the elements are contained. The term $\frac{1}{n}$ is the soundness error for the KKW online phase, we add the soundness error of the permutation proof as the term $\frac{\ell}{q-1}$, lastly, we subtract the term $\frac{\ell}{q-1}\frac{1}{n}$, which is the probability of both soundness errors occurring. Thus, we minimize the number of online repetitions $\tau$ such that

$$\epsilon(m,n,\tau) \stackrel{\text{def}}{=} \max_{m-\tau \leq k \leq m} \left\{ \frac{\binom{k}{m-\tau}}{\binom{m}{m-\tau} \cdot \left( \frac{(q-1)n}{q-1+n\ell-\ell} \right)^{m-\tau-k}} \right\}$$

is $\leq 2^{-128}$ for different values of $n$. For a list size $\ell = 2^{16}$ and a field size $q = 2^{64}$, the soundness error that we have added by introducing the permutation proof gets fully reduced in the same number of rounds that are needed to amplify the soundness error introduced by KKW. Hence, no extra repetitions are needed for adding the unknown permutation proof inside a KKW protocol. Sample satisfying values can be found in Table 4.

**Using a Ring Instead of a Field.** It is much easier to efficiently implement our scheme over a ring of size $q = 2^t$, for some $t \in \mathbb{N}$. However, the original Schwartz-Zippel lemma only works for polynomials over a field. As a general optimization, we work over the ring $\mathbb{Z}_{2^n}$, where we choose $n$ to be 64. Fortunately, a more general

| $\rho = 128$ | | | | | | |
|---|---|---|---|---|---|---|
| $n$ | 4 | 8 | 16 | 32 | 64 | 128 |
| $m$ | 218 | 293 | 352 | 606 | 842 | 1291 |
| $\tau$ | 65 | 43 | 33 | 26 | 22 | 19 |

**Table 4: Sample values for the number of preprocessing repetitions $m$, players $n$, and online repetitions $\tau$, similar to the values in KKW [33].**

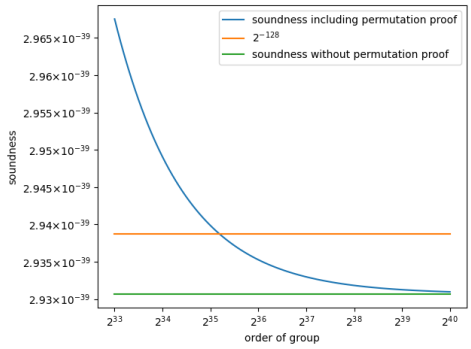


**Figure 3: Soundness error with permutation proof for 64 players, list of size $\ell = 2^{16}$, number of preprocessing runs $m = 842$, and online runs $\tau = 22$.**

form of the Schwartz-Zippel lemma, introduced by Arvind et al. [3], can be applied in this setting. They show that the Schwartz-Zippel lemma still holds when the random assignment to the polynomial is chosen within a finite subset of an integral domain contained within the ring. Within $\mathbb{Z}_{2^n}$ all odd numbers form such integral domain.

To make sure that our permutation proof still holds, the verifier needs to pick the challenge to be an odd number. The impact on the soundness is that instead of using the size of the overall ring, we have to use the size of the integral domain. This hardly impacts the soundness as long as the ring size is still large enough, which is the case for $\mathbb{Z}_{2^{64}}$.

In Figure 3 we show the impact of the arithmetic ring size on the soundness error, given a list size of $\ell = 2^{16}$ and number of players $n = 64$. We choose the number of preprocessing repetitions $m = 842$ and online repetitions $\tau = 22$. Based on this figure we can conclude that the most optimal ring size is $2^{36}$, but for ease of implementation we've implemented a ring of size $2^{64}$.