HAWKEYE – Recovering Symmetric Cryptography From Hardware Circuits

Gregor Leander[1], Christof Paar[2], Julian Speith[2], and Lukas Stennes[1]

[1] Ruhr University Bochum, Bochum, Germany
{gregor.leander,lukas.stennes}@rub.de
[2] Max Planck Institute for Security and Privacy (MPI-SP), Bochum, Germany
{christof.paar,julian.speith}@mpi-sp.org

Abstract. We present the first comprehensive approach for detecting and analyzing symmetric cryptographic primitives in gate-level descriptions of hardware. To capture both ASICs and FPGAs, we model the hardware as a directed graph, where gates become nodes and wires become edges. For modern chips, those graphs can easily consist of hundreds of thousands of nodes. More abstractly, we find subgraphs corresponding to cryptographic primitives in a potentially huge graph, the *sea-of-gates*, describing an entire chip. As we are particularly interested in unknown cryptographic algorithms, we cannot rely on searching for known parts such as S-boxes or round constants. Instead, we are looking for parts of the chip that *perform highly local computations*. A major result of our work is that many symmetric algorithms can be reliably located and sometimes even identified by our approach, which we call HAWKEYE. Our findings are verified by extensive experimental results, which involve SPN, ARX, Feistel, and LFSR-based ciphers implemented for both FPGAs and ASICs. We demonstrate the real-world applicability of HAWKEYE by evaluating it on OpenTitan's Earl Grey chip, an open-source secure micro-controller design. HAWKEYE locates all major cryptographic primitives present in the netlist comprising 424 341 gates in 44.3 seconds.

Keywords: Hardware Reverse Engineering · Symmetric Cryptography

1 Introduction

Proprietary and non-public cryptography is both more common than one would expect and as harmful as expected in almost all cases. Recent examples of disclosed and afterward broken non-public cryptography include GEA-1, an intentionally weakened cipher for mobile communication [9], and the TETRA ciphers, some of which are again backdoored [46]. A certainly not exhaustive list of other examples includes the attacks on DST40 [19], KEELOQ [32], the Mifare Crypto-1 cipher [56], and the SimonsVoss digital locking system [64].

Uncovering such secret or proprietary cryptography in an application, however, is a difficult task. In fact, many of the aforementioned insecure ciphers had been used for *decades* before the algorithms and their weaknesses were discovered. This exposes the public to many threats by, e.g., intelligence agencies.

To counter such threats, tools to automate the task of finding cryptography in software have recently been proposed. For instance, the work of Meijer et al. [47] assists during the software reverse engineering process and allows locating known and unknown cryptographic primitives in software binaries. For hardware, the situation is very different. In their work uncovering the TETRA ciphers [46], Meijer et al. state that, although they were confident that the integrated circuit (IC) they identified contained TETRA in hardware, they deliberately decided against hardware reverse engineering as it «is considerably more time and resource intensive» than software reverse engineering. For TETRA, they got lucky as software implementations of the cipher existed, hence, they could simply resort to software reverse engineering instead. However, particularly for special-purpose, proprietary ciphers, this is usually not the case. That is, there are cases where hardware reverse engineering is the only available option.

"Hardware" today exists in two principal forms: application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs). While the functionality of an ASIC is fixed during manufacturing, FPGAs can be programmed to implement almost any function using a configuration file, i.e., the *bitstream*, even after manufacturing. An application domain where (i) proprietary ciphers that are (ii) implemented in hardware are very common is in defense systems. In defense systems, FPGAs are often used as they are more cost-efficient in low-volume applications. The ongoing conflict in Ukraine gives two striking examples of such uses of FPGAs: An automatic targeting system recovered from a downed Russian SU-24M fighter jet [52] and the Russian R-187P1 Azart handheld radio transceivers [27]. Inside of both systems, Xilinx FPGAs were found. In the case of the radio, the FPGA supposedly contains (proprietary) ciphers. It is quite obvious that understanding such ciphers is of major interest, not only to the parties involved in the conflict but also to academics scrutinizing real-world deployments of cryptography.

Besides defense systems, other reasons are at least as important from a societal point of view. A general motivation is digital sovereignty, i.e., the desire of nation-states to minimize their dependency on foreign, untrusted hardware, e.g., in critical infrastructure components. A recent example with major political ramifications is the ban of certain suppliers of 5G infrastructure hardware by the UK and the USA over the fear of hidden hardware backdoors. Such mistrust is not entirely unfounded, considering, e.g., the case of the Crypto AG [48] or the allegedly «SIGINT enabled» CPU vendor Cavium [4]. The massive political and economic implications of digital sovereignty become evident when looking at the EU Chips Act and the US CHIPS and Science Act, both of which plan to invest 10s of *billions* of Dollars in order to reduce the reliance on foreign hardware manufacturers. In fact, the former is one of the largest subsidizing projects in the history of the European Union. Against this background and given the numerous examples of weak or even backdoored cryptographic algorithms, we argue that locating and analyzing cryptographic primitives in hardware is an important desideratum for the scientific community.

An early example of this line of work comes from Nohl et al. [56], who recovered and attacked the cipher implemented on the Mifare Classic RFID tag by reverse engineering the respective chip. Hardware reverse engineering consists of two main stages: (i) netlist extraction and (ii) netlist analysis [5]. For the former, Nohl et al. used a mixed chemical-mechanical method to recover the chip layers. This step was followed by optical imaging and image processing, resulting in the chip's netlist, a collection of logic gates and memory elements and their interconnections. Since the netlist consisted of only 400 gates, the second phase—netlist analysis—could be performed manually.

Today's digital ICs commonly consist of at least a few 100,000 gates and chips in high-end products such as smartphones and laptops often contain 100s of millions of gates. Hence, manual approaches as used for the Mifare Classic have become infeasible. For ASICs, the netlist extraction process is laborious due to shrinking transistor sizes [41,61,71], but well understood. In fact, besides specialized agencies of nation-states, several commercial service providers offer netlist recovery [68,69]. For FPGAs, netlist extraction is even easier to achieve and can be done with moderate resources, e.g., by academics. Here, recovering a netlist merely requires reverse engineering the proprietary file format of the bitstream, which is then converted into a gate-level netlist. This process is well understood and documented [36,73,57,12,28,59,3]. However, we would like to stress that in both the ASIC and FPGA case, a reverse engineer who has recovered the netlist is faced with a large sea of gates that lacks any hierarchy, module boundaries, or even just labels, data types, or meaningful strings often found in software binaries [40]. Given that modern hardware implementations can feature many millions of logic gates, analyzing such a netlist is extremely difficult without automation. Some specialized approaches towards automatic gate-level netlist analysis have been proposed so far [45,62,21,5,2]. Surprisingly, despite initial experiments [72,2,65], no approach toward the automated detection and identification of cryptography in gate-level netlists exists.

Our Contribution. The work at hand aims to address this research gap by proposing a set of algorithms to locate and analyze symmetric cryptographic primitives within gate-level netlists, see Fig. 1. We refer to this collection of algorithms as HAWKEYE.[3] Our approach allows us to automatically detect symmetric primitives such as block ciphers, hash functions, or permutation-based cryptography. Most importantly, our techniques allow us to recover known algorithms and unknown—potentially secret or proprietary primitives—from hardware circuits. To this end, we leverage the observation that implementations of such algorithms result in structural properties that are inherent in symmetric ciphers. More specifically, symmetric ciphers operate very locally, i.e., repeatedly applying the same round function to their state.

[3] HAWKEYE—**Ha**rd**w**are recovery of un**k**nown symm**e**tric cr**y**ptographic impl**e**mentations. Metaphorically speaking, HAWKEYE circles over the sea of gates and dead on target finds its prey: symmetric cryptography.

HAWKEYE comprises three steps that can be executed independently of each other. First, (i) it interprets the sea of gates as a graph and traverses this graph to identify gates likely belonging to a cryptographic implementation. Second, (ii) HAWKEYE extracts a description of the cryptographic primitive's round function and annotates external inputs such as state, plaintext, round key, and control. Finally, (iii) it analyzes the round function. For substitution-permutation network (SPN) ciphers, we attempt to automatically extract S-boxes and match them against a database of known S-boxes. The extracted S-boxes can also be exported for further analysis using tools such as `sagemath` [70] or `SboxU` [8].

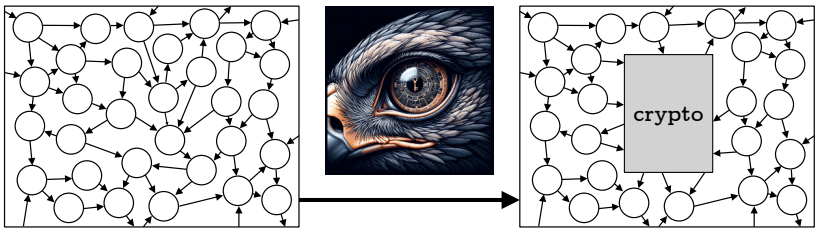


Fig. 1: Illustration of HAWKEYE locating cryptography in the sea-of-gates.

For the evaluation of HAWKEYE, we rely on open-source hardware designs. We synthesized these designs for FPGAs and some also for ASICs to obtain gate-level netlists. In our first experiment, which is close to a real-world setting, we executed HAWKEYE on an FPGA netlist implementing OpenTitan's Early Grey secure micro-controller that comprises 424 341 gates. HAWKEYE reliably located all major cryptographic primitives implemented on Early Grey, including AES [26], Keccak [14], and SHA-256 [55] in 44.3 seconds. Furthermore, it correctly extracted and identified the AES S-boxes. Given the circuit size, we assume manual approaches for finding the ciphers would have been extremely time-consuming.

In order to evaluate HAWKEYE with a large number of symmetric algorithms, we used a relatively small system-on-chip (SoC) synthesized with varying cryptographic co-processors. We simulated a realistic detection setting by surrounding the cipher with non-cryptographic circuitry. A main finding of these evaluations is that the structural properties HAWKEYE exploits for detection are actually present across different ciphers and independent of surrounding circuitry. We correctly recovered the state and/or (round-)key registers of 17 out of 18 FPGA benchmarks and all 18 ASIC benchmarks, often in a fraction of a second. As a cryptographic implementation usually revolves around its state register(s), this gives us the location of the respective cipher on the chip. As there are, by nature, no public hardware designs of non-public cryptographic algorithms, we relied on open-source implementations of known ciphers for this evaluation. Given that we detected all tested public algorithms, we are confident that proprietary ciphers would also be identified by HAWKEYE.

As a final set of experiments, we examined the false positive rate of HAWKEYE. We applied it to six non-cryptographic implementations, where only between zero and three false candidates were reported. A surprising result is that HAWKEYE also identified some LFSR-based PRNGs, even though it has been primarily designed to locate SPN, Feistel, and add-rotate-XOR (ARX) ciphers.

HAWKEYE is implemented as a plugin for the open-source hardware analyzer HAL [33,37] and is available as part of the HAL GitHub repository.[4] It is designed to be easy to use as it only requires a few straightforward function calls and does *not* rely on fine-tuning any parameters to make it work. Still, if known, a user can specify, e.g., a block size or S-box entries that they suspect to find.

Finally, we want to note that our approach is not designed to deal with implementations protected against side-channel or fault attacks. Of course, tackling protected implementations is appealing for future work. As HAWKEYE is built in a modular way, we hope that future techniques will be integrated into our tool.

Related Work. For a general overview of hardware reverse engineering and netlist analysis in particular, we refer to the work of Azriel et al. [5]. Experiments of Werner et al. [72] discuss graph partitioning in the context of reverse engineering cryptographic cores. Their approach produces a fuzzy partition and requires manual interpretation of the resulting partitions based on known patterns. Albartus et al. introduced DANA [2] as a plugin for HAL that attempts to recover high-level registers from a gate-level netlist. One of their case studies examines an early version of the OpenTitan chip and reveals its registers, including those of an AES and SHA-256. Still, locating and identifying the implemented ciphers requires manual inspection of the resulting register graph. Furthermore, Swierczynski et al. [65] induce permanent faults into AES and DES implementations on FPGAs, particularly their S-boxes, with the goal to recover key material. They argue that the parts of the FPGA configuration exhibiting high non-linearity are likely to implement an S-box. These initial studies provide some insights into detection strategies for cryptographic implementations, but none of them presents a comprehensive solution.

For software, Meijer et al. presented *Where's Crypto?* [47] at USENIX 2021. The goals of our work are aligned with theirs, but while they presented solutions to find and identify cryptography in software, we are concerned with hardware implementations instead. Their tool relies on predefined signatures of the cryptographic implementations and then on solving graph isomorphism problems to identify those signatures in binaries. As the challenges associated with hardware reverse engineering are fundamentally different from software reverse engineering, their approaches cannot simply be transferred to the hardware domain.

In case we detect an S-box in the sea-of-gates, we make use of a technique proposed by Biryukov et al. [15] to efficiently check whether the S-box is present (up to affine equivalence) in a database of well-known S-boxes. Of course, in case HAWKEYE comes across unknown cryptographic building blocks, further analysis

[4] See https://github.com/emsec/hal/tree/8f6c18d4473ef2bdc80f0128b78bab16a4b99fea/plugins/hawkeye.

is needed. While not in the scope of our work, we also want to mention further works of Biryukov et al. [16] and Beierle et al. [10], who reverse engineered S-boxes and linear layers to uncover hidden structures.

Outline. We recall relevant preliminaries in Section 2. In Section 3, we explain our techniques to locate and analyze cryptographic implementations in netlists. After that, in Section 4, we evaluate the resulting tool on OpenTitan's Earl Grey secure micro-controller, on a smaller SoC with cryptographic accelerators, and numerous standalone implementations of symmetric cryptographic algorithms. We conclude our work in Section 5.

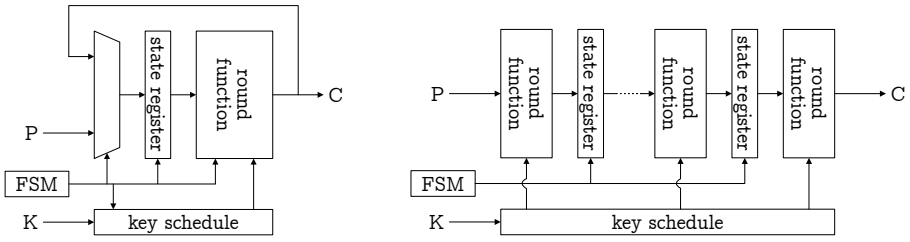
2 Preliminaries

While symmetric cryptography can, in a top-down approach, be categorized into different groups like encryption, hash function, and message authentication codes which again can be divided into block ciphers, stream ciphers, and so on, virtually all of them follow the same principle: Update some state repeatedly with an efficient round function to derive a random-looking state. For security, it is crucial that the round function features *non-linearity*. The most common building blocks to achieve this are S-boxes as the heart of SPNs, non-linear filter or update functions for (non-)linear-feedback shift registers, and modular additions in ARX designs.

2.1 Hardware Implementations of Symmetric Ciphers

Generally, cryptographic algorithms are implemented in hardware, i.e., FPGAs or ASICs, using sequential and combinational logic. Sequential logic such as flip-flops holds the current state of the circuit and is updated on a clock signal's positive (or negative) edge. Hence, it synchronizes the circuit. Such flip-flops may be controlled by inputs such as *enable*, *reset*, or *set*. Combinational logic comprises logic gates that compute a Boolean function on their inputs and immediately (or with minimal delay) output the result. Computations such as a round function of a symmetric cipher are implemented in combinational logic, while sequential logic stores (and synchronizes) intermediate states.

Different approaches can be pursued when implementing symmetric cryptography. Here, we focus on typical implementation approaches for SPN, Feistel, and ARX-based ciphers. Round-based implementations as depicted in Fig. 2a compute one round of a cipher per clock cycle and store the intermediate state after this round in the *state register* that consists of flip-flops. In our example, the state register input is connected to a multiplexer that selects the plaintext upon initialization and the current state while the later rounds are computed. Hence, round-based implementations are compact because the round function needs to be implemented only once, but they lack throughput as multiple clock cycles are required to compute a single ciphertext block. The complex inner workings of such an implementation are typically steered by a finite state machine (FSM).



(a) Round-based implementation. (b) Pipelined implementation.

Fig. 2: Common implementation techniques for symmetric ciphers.

Pipelined implementations such as shown in Fig. 2b present the other extreme. Here, the implementation is unrolled, i.e., every round of the cipher is implemented separately using combinational logic, interrupted only by state registers. The state registers are placed between the rounds to reduce the critical path length, which is required to achieve high clock frequencies. Generally, this setup allows multiple different plaintexts to be at different stages of the encryption at the same time. If fully utilized, one ciphertext is output every clock cycle, drastically improving throughput at the cost of more logic area being required for the implementation.

While these two approaches are most common, other implementation techniques exist. For instance, the cipher PRINCE is built in a way that allows it to be implemented fully unrolled [20], without any state register between the individual rounds. Furthermore, highly area-optimized implementations may also deviate from these patterns. Fig. 2 shows examples of round-based and pipelined implementations, but different flavors of them exist. The state register of a round-based implementation may also be placed behind the round function so that the combinational logic of the multiplexer and the round function can be merged. For pipelined implementations, more than one round may be implemented between two state registers to reduce latency. Finally, additional plaintext, ciphertext, and key registers may be present at the inputs and outputs of such implementations. Even such slight deviations already present challenges to a generic detection algorithm.

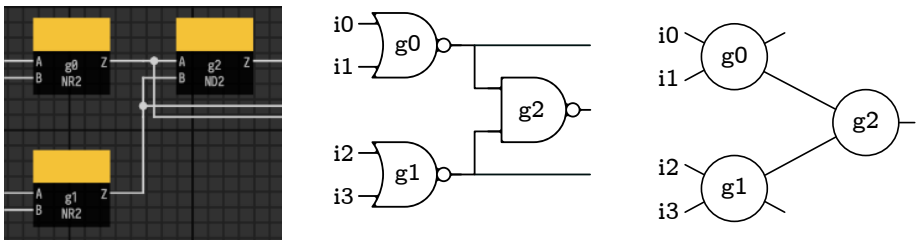
2.2 Hardware (Reverse) Engineering

Hardware is built by first describing the desired components in a hardware description languages such as Verilog. This description is then fed into a synthesizer to obtain a gate-level netlist. Such a netlist comprises the combinational gates and sequential elements such as flip-flops of a hardware implementation, as well as their interconnections, i.e., the nets connecting the gates. The primary difference between ASIC and FPGA netlists is that combinational logic is realized using look-up tables (LUTs) on FPGAs while ASICs rely on regular Boolean

gates. For ASICs, the netlist is then implemented as a layout file describing the physical properties of the final chip. Afterward, the layout file is passed on to a manufacturing facility that actually produces the chip. In contrast, FPGAs feature an already manufactured (and hence fixed) fabric that can be programmed to implement arbitrary, user-defined circuits by using a configuration file also referred to as a *bitstream*. To obtain this bitstream, the gate-level netlist is placed and routed on the existing fabric using an electronic design automation tool provided by the FPGA vendor. Afterward, it is converted into the proprietary bitstream format of the respective FPGA.

For simplification and in line with previous work [5], we consider hardware reverse engineering to be a two-stage process. First, during *netlist extraction*, a gate-level netlist is recovered from a target, either an FPGA or an ASIC. For ASICs, this requires skilled personnel, expensive equipment, weeks or months to complete, and is prone to errors [41,61,71]. However, for FPGAs, netlist extraction merely involved reverse engineering the format of their *bitstream* file, which is a well-understood process [36,73,57,12,28,59,3]. While bitstream protections such as encryption exist, they have repeatedly been shown to be susceptible to protocol [35,34] and side-channel attacks [49,50,51,66,67]. On top of this, recovery of an error-free netlist is achievable given a sound understanding of the bitstream format. Recent studies have shown reverse engineering of an entire real-world FPGA for the purpose of intellectual property theft to be feasible [40].

Second, having recovered a gate-level netlist, *netlist analysis* is performed to achieve a high-level understanding of the implementation. The exact meaning of this depends on the goals of the reverse engineer. The work at hand deals only with this second stage of hardware reverse engineering in the context of locating and analyzing implementations of arbitrary symmetric ciphers in unknown gate-level netlists implemented on FPGAs or ASICs. To this end, we are confronted with a *sea-of-gates* that lacks obvious structure such as hierarchy and module boundaries, or even just interpretable labels or word-level information such as data types and bit orders often used for software reverse engineering [40].



(a) A small gate-level netlist in HAL.

(b) The same netlist as a circuit diagram.

(c) The same netlist interpreted as a graph.

Fig. 3: An excerpt from a ASIC netlist comprising three combinational gates shown in different forms of representation.

Fig. 3 depicts a small excerpt from a gate-level netlist extracted from an ASIC in the netlist analysis framework HAL [33,37] (Fig. 3a) and as a circuit diagram (Fig. 3b). In our work, we interpret the netlist as a graph $G = (V, E)$, see Fig. 3c. To this end, we treat the gates in the netlist as vertices V and the nets as directed edges E. For some vertices $v_0, v_1, v_2, \ldots, v_n \in V$, we say that $p = (v_0, v_1, \ldots, v_n)$ is a *path* of length $|p| = n$ if $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$. Notice that we do *not* require that the v_i are distinct. In case we do not care about the inner vertices, we simply write $v_0 \overset{p}{\rightsquigarrow} v_n$. The set of all paths connecting v_0 and v_n is denoted by $P(v_0, v_n)$.

To conduct netlist analysis, both *structural* and *functional* methods can be applied. Here, structural means that we ignore the Boolean functions implemented by the combinational gates. For functional analysis, these Boolean functions are considered by labeling nodes representing combinational gates with the Boolean function they implement. We split gates with multiple outputs into single-output gates. Based on these labels, we evaluate subgraphs to learn the combined Boolean functions of multiple gates. For instance, in Fig. 3, the output of vertex `g2` as a function of the inputs of vertices `g0` and `g1` is

$$f(i_0, i_1, i_2, i_3) = \neg(\neg(i_0 \vee i_1) \wedge \neg(i_2 \vee i_3)) = i_0 \vee i_1 \vee i_2 \vee i_3.$$

For performance reasons, structural analysis is preferable over functional analysis, as the traversal of the netlist graph is usually faster than operating on complex Boolean functions comprising dozens of input variables.

3 Our Techniques

In this section, we explain the technical details and design choices of HAWKEYE. For illustration, we use a round-based AES hardware implementation on a Xilinx 7-series FPGA as a running example. We deliberately omit non-instructive details and use a rather theoretical perspective, s.t. no deep knowledge of hardware implementations is needed to follow our description. Of course, the interested reader is invited to consult our implementation of HAWKEYE for details.

On a high level, HAWKEYE comprises three independent steps: (i) locating candidates of cryptographic implementations, (ii) extracting and dissecting the round function, and (iii) analyzing the round function. Step (i) takes the netlist graph G as input and outputs candidates that HAWKEYE suspects to be state registers of a cryptographic implementation. More formally, this step outputs a set $\{C_1, C_2, \ldots\}$ of candidates $C_i = (R_{i,\text{in}}, R_{i,\text{out}})$. Each C_i comprises two sets of flip-flops where one set of flip-flops represents the input register $R_{i,\text{in}}$ and the other the output register $R_{i,\text{out}}$. Steps (ii) and (iii) are then executed on all candidates separately.

Step (ii) takes one candidate C_i from Step (i) as input and then outputs a graph representing only the round function which exactly contains the previously identified input and output registers, the combinational logic in between these two registers, and additional inputs like round keys. While appearing superfluous

at first, this step acts as a layer of normalization and enables us to prepare the round function for further analysis.

In Step (iii), `HAWKEYE` analyses the extracted round function graph and, for SPNs, tries to recover S-boxes through structural and functional analysis. Furthermore, `HAWKEYE` attempts to identify these S-boxes by comparing them against a database of known S-boxes using the affine equivalence algorithm from [15].

3.1 Locating Cryptographic Candidates

The most important step of our analysis is the detection of (candidates of) cryptographic state registers. As this step is executed on the entire gate-level netlist, which can contain thousands if not millions of gates, we rely solely on *structural* techniques. That is, we only work on the graph induced by the netlist but do *not* evaluate any Boolean functions. Conveniently, the fact that symmetric primitives work extremely locally translates well to such a structural check.

To be more specific, consider round-based and pipelined implementations as described in Fig. 2. As stated before, we interpret the netlist as a directed graph $G = (V, E)$. Now, to locate cryptographic implementations, we consider only the *flip-flop connectivity graph* $G_{\mathrm{FF}} = (V_{\mathrm{FF}}, E_{\mathrm{FF}})$ implied by replacing all combinational gates with nets. That is, $V_{\mathrm{FF}} \subset V$ is the set of all flip-flops in V and E_{FF} is such that there is an edge (u, v) if there is a path from flip-flop u to v which passes only through combinational logic but not through other flip-flops.

For a moment, assume that no ciphertext register exists in G_{FF}. While this is not realistic in practice, it helps to understand our approach and we later show how to deal with this issue. If there is no ciphertext register, then every flip-flop in the state is followed only by other state flip-flops. In other words, there is no branching out of the state data path; therefore, the state register forms a sink. This observation is quite intuitive as it would be a security disaster if the state would influence anything but itself and the ciphertext. As this observation does not hold for most flip-flops that do not belong to a cryptographic implementation, it enables us to locate candidates of state registers within the sea-of-gates using only structural techniques.

More formally, the *k-th neighborhood* $N_k(v)$ of a state flip-flop v in G_{FF},

$$N_k(v) = \{u \in V_{\mathrm{FF}} \mid \exists p \text{ s.t. } v \overset{p}{\rightsquigarrow} u \text{ and } |p| = k\},$$

consists only of state flip-flops. Moreover, for strong symmetric primitives, the avalanche effect ensures that the process of considering $N_k(v)$ for increasing k quickly obtains a complete state register. That is, there is a small k' s.t. the k'-th neighborhood constitutes a complete state register and for $k' + 1$ we again get a complete state register. We depict this for a toy example in Fig. 4a using a state size of four. For our running example, i.e., the round-based implementation of AES, we have $k' = 2$ as two rounds of AES provide full diffusion.

Based on this idea, we assemble Algorithm 1. It takes the flip-flop connectivity graph G_{FF} derived as described above as input. Then, for all flip-flops, we compute the k-th neighborhoods where $k = 1, 2, \ldots, k_{\max}$. As we expect to

identify state registers with a small k, we choose $k_{\max} := 10$. If we encounter a flip-flop v for which the size of the neighborhoods quickly saturates and is larger than a heuristically predefined minimal register size of $c_{\min} = 10$, we stop and store the two last neighborhoods as the two registers R_{in} and R_{out} of a candidate C. We might encounter this candidate multiple times for different start flip-flops v, but we only output it once. Also, we might detect a candidate $C = (R_{\text{in}}, R_{\text{out}})$ which is a superset of another candidate $C' = (R'_{\text{in}}, R'_{\text{out}})$, in the sense that both $R'_{\text{in}} \subsetneq R_{\text{in}}$ and $R'_{\text{out}} \subsetneq R_{\text{out}}$. For instance, this might happen if we additionally start in a key schedule flip-flop and thereby find a candidate that consists of the round key register and the state register. If C is a superset of C', we discard C and output only C'. Finally, we only consider candidates containing at least $s_{\min} = 40$ flip-flops in their output register to be state register candidates.

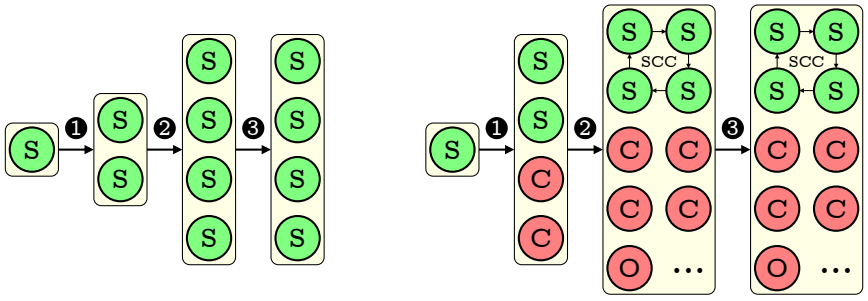
Algorithm 1 Identify state registers.

Require: flip-flop connectivity graph G_{FF}
Ensure: state-register candidates
  $k_{\max} := 10$ ▷ maximal number of forward steps
  $c_{\min} := 10$ ▷ minimal candidate size
  $s_{\min} := 40$ ▷ minimal state register size
  for each $v \in V_{\text{FF}}$ **do** ▷ for all flip-flops
    for $k = 1$ **to** $k_{\max}$ **do**
      Compute k-th neighborhood $N_k(v)$ ▷ using breadth-first search
      if $c_{\min} \leq |N_{k-1}(v)| = |N_k(v)|$ **then**
        Store $(N_{k-1}(v), N_k(v))$ as a candidate for input and output register
        break
      end if
    end for
  end for
  return candidates that are larger than $s_{\min}$ and not a superset of another candidate

As Algorithm 1 relies on the heuristics described above, claiming any theoretical correctness is inappropriate. Of course, if a cryptographic implementation is not covered by our heuristic, Algorithm 1 can not detect it. Hence, we instead give practical results in Section 4 for detection rates and runtime complexity. Regarding runtime, the most expensive step is the breadth-first search to compute the neighborhoods, which we execute for at most $k_{\max} = 10$ steps for every flip-flop in the netlist.

Recall that we have assumed that no ciphertext register exists in G_{FF} at the beginning of this section and that every state flip-flop would only be followed by other state flip-flops. This is true for pipelined implementations for all but the last round, since every state register is only followed by another state register. However, as is, Algorithm 1 fails for round-based implementations as the one existing state register does not only feed back into itself but also into the ciphertext register. Hence, when traversing G_{FF} starting from a state flip-flop, we need to ensure to exclude ciphertext flip-flops to prevent "exiting" the state logic. We

present two complementary methods to deal with this issue. As HAWKEYE is built in a modular way, further methods could easily be added over time if the need arises. Both presented methods can also be combined to refine results.



(a) Candidate search with Method 1. (b) Candidate search with Method 2.

Fig. 4: Different approaches to locate the state register of an implemented cipher based on forward propagation. State flip-flops are denoted as (S), ciphertext flip-flops as (C), and other flip-flops as (O).

Method 1: Flip-Flop Filtering. On FPGAs, we observed that the state and ciphertext registers are often controlled by distinct control inputs directly applied to the flip-flops. We utilize this observation to refine our candidate search by applying an approach we refer to as *flip-flop filtering*. In case V_{FF} contains flip-flops with different control inputs, we consider $G^f_{\mathrm{FF}} = (V_{\mathrm{FF}}, E^f_{\mathrm{FF}})$ with E^f_{FF} containing $(u, v) \in E_{\mathrm{FF}}$ only if both u and v are controlled by the same control inputs. We then execute Algorithm 1 on G^f_{FF} instead of G_{FF}. Our experimental evaluation in Section 4 demonstrates that this usually produces good results for FPGAs, but not so much for ASICs.

Example 1 (Round-based AES). To illustrate our techniques, we apply them to a round-based implementation of AES on a Xilinx 7-series FPGA and report not only the results but also describe the inner workings. We choose AES as it is the best-known symmetric cipher and its implementation, especially on FPGAs, is an exceptionally instructive example. Apart from implementing AES, the netlist under scrutiny also comes with a Universal Asynchronous Receiver Transmitter (UART) interface. This allows us to communicate with an FPGA board configured with the netlist so that we can actually encrypt plaintexts and obtain ciphertexts using our implementation. The sole purpose is to demonstrate that our AES implementation is indeed correct.

Executing HAWKEYE on the netlist consisting of 3458 gates takes only a fraction of a second and produces one candidate. Indeed, the candidate consists of two identical registers comprising the same 128 flip-flops each and corresponds to the

AES state register. To better understand how HAWKEYE behaves during candidate search, we dissect its inner working for several start flip-flops from our example: A state flip-flop, flip-flops from the key schedule, the plaintext, the FSM steering the AES implementation, and several flip-flops from the UART interface. For each of them, we give the sizes of the neighborhoods $N_k(v)$ in Table 1.

Table 1: Exemplary traces of our candidate search for different start flip-flops.

i	**Register of v_i**	**($\lvert N_k(v_i)\rvert$ for $k = 1, 2, \ldots$)**	**Candidate**	**Comment**
1	state	(32, 128, 128)	✓	
2	round key	(36, 164, 196, 228, 256, 256)	(✓)	superset of 1
3	plaintext	(0)	✗	different control
4	AES state machine	(4, 4)	✗	
5	UART state machine	(2, 2)	✗	
6	UART clock counter	(10, 10)	✗	
7	UART transmitter	(0)	✗	different control
8	UART receiver	(0)	✗	different control

For the state flip-flop v_1, we get $|N_1(v_1)| = 32$, i.e., v_1 influences 32 other flip-flops. With AES in mind, this is the expected behavior since the signal passes through one 8-bit S-box and `MixColumns`. Next, we have $|N_2(v_1)| = 128$. Again, with the diffusion properties of AES in mind, this is expected as `ShiftRows` ensures that the flip-flops in $N_1(v_1)$ cover all four columns of the state and hence we collect all state flip-flops in $N_2(v_1)$. Now, we have $|N_3(v_1)| = 128$, simply because the complete state influences all of the state again. This also implies that $|N_k(v_1)| = 128$ for all $k \geq 2$. Hence, we stop, and the state register is correctly identified as a candidate.

For most other flip-flops, we can also stop the search after only a few steps. This is either because the respective flip-flop has no successors in G^f_{FF} due to differing control inputs or because we quickly saturate at a neighborhood smaller than $c_{\min}$. Note that for v_6, a candidate containing 10 flip-flops is created, but not considered a state register candidate as it is smaller than $s_{\min}$. One exception is the round-key register, for which we end up with a candidate containing both the state and the round-key register. This candidate constitutes a superset of the identified state candidate and is discarded.

Method 2: SCC Detection. For round-based implementations on ASICs, we observed that distinguishing state and ciphertext registers based on their control inputs is not always possible. Some control inputs are not directly applied to the flip-flops but merged into the combinational logic preceding a flip-flop's data input. Here, recovering the control inputs feeding into the combinational logic is sometimes impossible without making assumptions about the surrounding circuitry. However, due to the nature of symmetric cryptography, the state register flip-flops in G_{FF} of such round-based implementations are usually strongly

connected. That is, eventually, every state flip-flop influences every other state flip-flop. In anticipation of the strongly connected components (SCCs) within G_{FF} being rather large and containing way more than just the state register, we implement a local SCC search within Algorithm 1 that only tries to find SCCs within the neighborhoods $N_k(v)$. Furthermore, we modify Algorithm 1 to only stop if the size of the SCCs found in $N_k(v)$ and $N_{k+1}(v)$ is equal. Only the flip-flops contained in the SCC are then considered to be the candidate. This method is depicted in Fig. 4b. As ciphertext flip-flops can no longer be differentiated from the state flip-flops, they are added to the set of currently considered flip-flops through graph traversal. Furthermore, other flip-flops may be added to this set in subsequent iterations by following the ciphertext flip-flops. However, as they are not part of the SCC, they will not be considered for the final candidate. We stress that this approach is not useful to find pipelined cryptographic implementations, because these do not form an SCC.

3.2 Extracting and Dissecting the Round Function

Once a candidate $C = (R_{\mathrm{in}}, R_{\mathrm{out}})$ has been found, `HAWKEYE` attempts to extract the round function implemented in combinational logic between R_{in} and R_{out}. For this, we again consider the complete netlist graph G. If $R_{\mathrm{in}} = R_{\mathrm{out}}$, we copy R_{in} and replace R_{out} with this copy s.t. we always end up with a directed acyclic graph. While cutting out the combinational subgraph between two registers appears straightforward, there are intricate details to consider in practice.

First, there likely are additional inputs to the round function other than the previous state. Those typically include the plaintext, round keys/constants, and control signals coming from an FSM. Labeling those as accurately as possible is important since we need to brute force the control inputs later on. Luckily, control inputs can often be easily identified as they connect to many gates in the round function. For instance, in our AES example, a control signal determines whether the `MixColumns` step takes place (it is omitted in the last round of AES). This control signal feeds into 128 combinational gates as it must influence the computation of every state bit. In contrast, a plaintext or round key flip-flop usually only feeds into a single combinational gate. Hence, heuristically, we label an additional input as control if the number of gates it is connected to is larger than half the state size.

Second, we check for special structures like Feistel. For Feistel networks, parts of the current state are just forwarded to the next state, without any computations being performed. In the netlists, this translates to some flip-flops in the input register R_{in} being directly connected to flip-flops in R_{out}, without any combinational logic in between.

Finally, dissecting the round function into independent parts might be possible. By considering the connected components of the undirected version of the flip-flop graph $G_{\mathtt{FF}}$, i.e. the graph $G'_{\mathrm{FF}} = (V'_{\mathrm{FF}}, E'_{\mathrm{FF}})$ with

$$V'_{\mathrm{FF}} = V_{\mathrm{FF}} \quad \text{and} \quad E'_{\mathrm{FF}} = \{\{u, v\} \mid (u, v) \in E_{\mathrm{FF}}\},$$

we identify independent parts of the round functions. Of course, it might be the case that the entire round function is just a single large component. However, considering one round of AES, we end up with four independent components that correspond to the four columns of the AES state. We note that AES influenced the design of many modern ciphers, which hence incorporate a similar structure. Furthermore, the quarter-round pattern of ChaCha [13] also matches this structure. In case HAWKEYE detects such a structure, the subgraph implementing the round function is divided accordingly.

Example 2 (Round-based AES). In the round function extraction step, we start at every flip-flop of the output register and propagate backward until we encounter the flip-flops of the input register or a gate that is not influenced by the input register anymore, i.e., an additional input. As we deal with a round-based implementation, we replicate the state register to have separate input and output registers. Recall that the complete netlist consists of 3458 gates in total. The extracted round function consists of 128 state flip-flops (another 128 duplicated state flip-flops), 258 additional inputs, and 1328 LUTs.

First, we investigate the additional inputs. Using our heuristic from above, HAWKEYE identifies two of them as control inputs. Indeed, those two control whether a plaintext is loaded into the state and whether the MixColumns operation takes place. The other 256 inputs form the plaintext and the round key inputs. Finally, we detect the four independent columns, as discussed before.

3.3 Analyzing Substitution–Permutation Networks

Arguably, SPNs are the most common design strategy for symmetric primitives. In recent years, many hardware-friendly SPNs were proposed [11,18,20,38,30,14]. For SPNs, we are interested in their S-boxes. Hence, HAWKEYE attempts to automatically extract S-boxes from the round function and compare them to a database of widely known S-boxes. For efficiency, our analysis is again mostly structural and minimizes the number of required Boolean function evaluations.

For completeness, we want to mention another approach: the techniques used to break SASAS [17]. Biryukov et al. treated the round function as a black box oracle to find the S-box by evaluating carefully crafted inputs. As this contradicts our goal of minimizing the number of evaluations, we did not study such techniques further.

Locating S-box Candidates. For a moment, assume that the round function starts with the S-box layer, i.e., the state input flip-flops R_{in} are indeed the inputs of the S-boxes. Still, a priori it is unclear which input is connected to which S-box. To group the flip-flops, we use a similar approach as before to identify independent components. Hence, we investigate connected components of the underlying undirected graph, but this time we also consider the combinational logic. Furthermore, instead of considering the entire graph at once, we step-by-step consider the subgraphs induced by taking k steps forward from the input

flip-flops R_{in} through the combinational logic. Assume that the round function $\mathcal{L} \circ \mathcal{S}$ consists of an S-box layer $\mathcal{S}$ and a linear layer $\mathcal{L}$. If we cut the graph such that the first part implements $\mathcal{S}$, then the connected components of the undirected version of the graph correspond to the S-boxes.

More formally, for two nodes $u, v \in V$, we denote the set of all paths from u to v as $P(u \rightsquigarrow v)$. Then, we define $\texttt{lcp}(v)$ as the *length of the critical (i.e. longest) path* from any input flip-flop $u \in R_{\text{in}}$ in the subgraph to v, i.e.,

$$\texttt{lcp}(v) = \max_{u \in R_{\text{in}}} \max_{p \in P(u \rightsquigarrow v)} |p|.$$

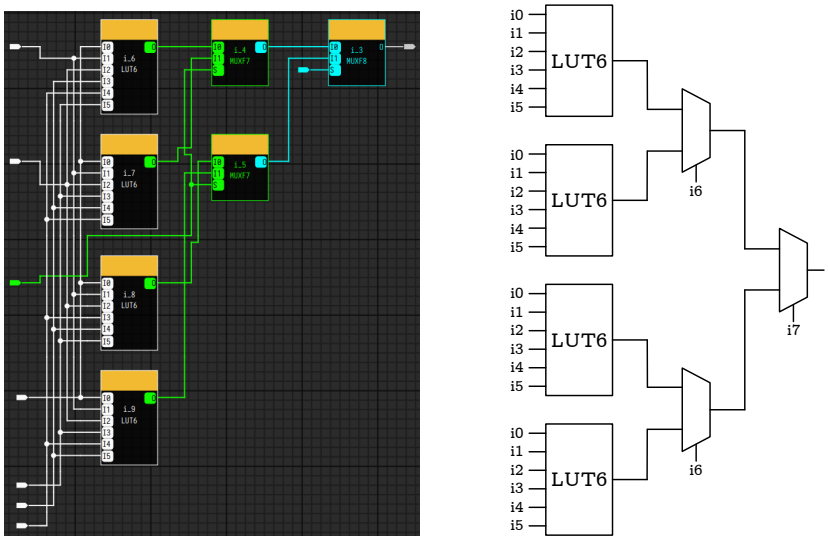
Then, in iteration k, we consider the graph induced by $\widetilde{V} = \{v \in V \mid \texttt{lcp}(v) \leq k\}$. If this graph *nicely* partitions into independent components, we have identified the S-box inputs. Here, of course, *nicely* conceals our heuristic, which is as follows: We check if there are $m \geq 1$ components, each containing exactly $n \geq 1$ input flip-flops such that $m \cdot n = |R_{\text{in}}|$ is the number of all input flip-flops.

Given the inputs of each S-box, we still have to determine the outputs. To this end, we first consider all gates that only depend on a non-strict subset of the inputs. Thereby, we are left with the gates that correspond to the computation of the S-box and potentially some linear operations after the S-box (think of multiplication with `0x03` in the `MixColumns` operation in the AES). We identify output gates by considering only the gates for which the successors also depend on inputs of other S-boxes. If we are left with the same number of outputs as inputs, we store the S-box candidate for further analysis. If we find fewer outputs than inputs, we discard the S-box candidate and continue with the next set of inputs or the next step. Of course, thereby we miss, e.g., the DES S-boxes. If we are left with more outputs than inputs, we assume this is due to optimizations of the linear layer that comes after the S-box. That is, in hardware, it might be beneficial to, e.g., compute an output bit of $\texttt{0x3}S(x)$ *directly* without using the outputs of $S(x)$. To address this, we check if there are outputs that depend only linearly on other outputs and, if so, eliminate them. Thereby we might obtain linear transformations of the S-box, but this is not a problem for `HAWKEYE` as both our identification of known S-boxes, as well as cryptographic properties of S-boxes are quite robust, i.e., invariant under affine equivalence.

Notice that so far we assumed that the round function is implemented as $\mathcal{L} \circ \mathcal{S}$. If it is $\mathcal{S} \circ \mathcal{L}$ instead, the inputs of a single S-box depend on more input flip-flops, and our technique as described above fails. However, we can apply the same idea backward. We start with the output flip-flops and walk, step-by-step, backward to find the S-boxes.

Example 3 (Round-based AES). Due to the nature of 7-series Xilinx FPGAs, every AES S-box is typically implemented the same way. To be more specific, 7-series FPGAs are organized in a grid of similar building blocks. Combinational and sequential logic are implemented in CLBs that are part of this grid and again split into two Slices each. Each Slice can implement any 8-input Boolean function from four 6-input LUTs whose outputs are connected by two layers of multiplexers. The resulting circuit for a single output bit is depicted in Fig. 5.

Returning to our S-box extraction, recall that we investigate each of the four identified components separately, i.e., the four columns of the AES state. For one, two, or three steps, our graph does not decompose *nicely* as there are input flip-flops that are not connected to anything else. These are the flip-flops that are directly connected to the last multiplexer in Fig. 5. When we go four steps forward, our graph decomposes into four components each consisting of eight input flip-flops followed by eight XORs for key addition and then eight instances of the subgraph shown in Fig. 5. For more than four steps, we find only a single component comprising all 32 flip-flops. In other words, we correctly located all input and output pairs of the S-boxes and there are no false positives.



(a) Implementation of one bit of the AES S-box in HAL.

(b) A circuit diagram of the same subcircuit.

Fig. 5: Implementation of one AES S-box output on a Xilinx 7-series FPGA.

Identifying S-Boxes. Once we locate candidates for the inputs and outputs of an S-box, it is simple to check whether the subcircuit in between them represents a known cryptographic S-box. First, we extract the Boolean functions for all output wires and combine them into a lookup table for the S-box. In case additional inputs contribute to the Boolean functions, we also have to assign values to them. To this end, we simply brute-force control inputs and execute all subsequent steps once for every possible control assignment. Furthermore, we fix all other non-state inputs (e.g., plaintext and round key) to zero.

Given the resulting lookup table, we apply a slight modification of the affine equivalence algorithm proposed by Biryukov et al. [15] to check whether the

S-box candidate is contained in a database of known S-boxes, see Algorithm 2. `HAWKEYE` incorporates such a database that can be extended by the user. For completeness, we point out that the work of Dinur [29] improves the affine equivalence algorithm in the case of *random functions*. Essentially, for the improvement to work, the investigated functions must be of a high degree. This is not always the case for S-boxes (e.g., consider the ASCON S-box of degree 2). Hence, we opted for the algorithm by Biryukov et al. [15], which is fast enough in our application. By relying on the *affine* equivalence algorithm, we ensure that our S-box extraction is rather robust. That is, the order of the inputs and outputs does not matter, nor does the addition of constants before or after the S-box. Even if the S-box and the linear layer are interwoven (think of the aforementioned multiplication of `0x3` with the AES S-box output), we identify the S-box correctly. Of course, this robustness also implies that we cannot distinguish affine equivalent S-boxes. Further, if the S-box is unknown, there are many equivalent solutions for decomposing the round function into S-box and linear layer, i.e., it is impossible to decide which part of the linear layer goes in the S-box.

Originally, the affine equivalence algorithm of Biryukov et al. [15] takes two S-boxes S and S' with $S, S' \colon \mathbb{F}_2^n \to \mathbb{F}_2^n$ as input and then decides whether those two S-boxes are affine equivalent, i.e., whether there exist affine and bijective maps $A, B \colon \mathbb{F}_2^n \to \mathbb{F}_2^n$ s.t. $S = A \circ S' \circ B$. Internally, for all choices of α and β, the affine equivalence algorithm computes and stores the *linear representatives* of $S(x) \oplus \beta$ and $S'(x \oplus \alpha)$, i.e., the lexicographical smallest S-boxes that are linear equivalent to $S(x) \oplus \beta$ and $S'(x \oplus \alpha)$ respectively. To check for affine equivalence, the algorithm computes the intersection of the linear representatives of S and S'. If there is a pair (α, β) such that the linear representatives match, then S and S' are affine equivalents. If we want to learn the name of an extracted S-box S, a trivial approach would be to run this algorithm repeatedly for a list of known S-boxes S'_i. For instance, we could first check whether an extracted 4-bit S-box is equivalent to the PRESENT S-box. If not, we check whether it is equivalent to the SKINNY S-box and so on. Hence, the algorithm would again and again compute the linear representatives for the known S-boxes. To avoid this, we slightly adapt the algorithm of Biryukov et al., see Algorithm 2. In an offline phase, we pre-compute the linear representative for our known S-boxes (for all choices of α) and store them in a database. In the online phase, we compute the linear representative of the unknown S-box S for all β and check whether it is contained in our database. In other words, this time-memory trade-off allows us to check against all known S-boxes in parallel instead of one by one.

The correctness and complexity of our algorithm follow from the original algorithm [15]. However, Biryukov et al. suggest that the `linear_representative` subroutine has a complexity of $O(n^3 2^n)$, which is not in line with our experiments. For example, for the AES S-box we obtain a worse complexity. But since we are only interested in rather small n (usually $n \leq 8$) and our implementation can handle those cases in fractions of a second, we did not investigate this incon-

sistency any further.[5] Our implementation is optimized by using large registers (AVX on x86 and NEON on ARM) to represent small sets frequently used in the the `linear_representative` subroutine. As we believe that our approach might also be interesting for other applications (e.g. software reverse engineering of cryptography), we will contribute our implementation of Algorithm 2 to `SboxU` [8].

Algorithm 2 Affine equivalence against database.

Require: List L of S-boxes and their *names* ▷ offline
Ensure: S-box database D
  for each $S' \in L$ **do** ▷ $S' \colon \mathbb{F}_2^n \to \mathbb{F}_2^n$
    for each $\alpha \in \mathbb{F}_2^n$ **do**
      $R \leftarrow$ `linear_representative`$(S'(x \oplus \alpha))$ ▷ lin. repr. algorithm from [15]
      $D[R] \leftarrow$ name of S' ▷ add to set of names if already defined
    end for
  end for
  return D

Require: S-box $S \colon \mathbb{F}_2^n \to \mathbb{F}_2^n$ and database D ▷ online
Ensure: Name of S-box S or $\perp$
  for each $\beta \in \mathbb{F}_2^n$ **do**
    $R \leftarrow$ `linear_representative`$(S(x) \oplus \beta))$ ▷ lin. repr. algorithm from [15]
    if $D[R]$ is defined **then**
      return $D[R]$ ▷ return the name(s) of the S-box
    end if
  end for
  return $\perp$ ▷ S is not in the database

In case Algorithm 2 fails, i.e., if the extracted S-box S is not contained in the database, we could apply standard analysis such as computing the differential uniformity, linearity, or algebraic degree. However, as these and many other properties of S-boxes are well-known, we argue that it is more reasonable to simply output such S-boxes and then rely on established tools such as `sagemath` [70] or `SboxU` [8] for further analysis.

Example 4 (Round-based AES). As established before, we correctly extracted all inputs and outputs of the sixteen AES S-boxes. Now, we compute the Boolean function for each output bit. For this, we actually need to assign a value to the eight round key bits that are xored to the inputs of the S-box. As we identified the key bits as additional non-control inputs, we simply set them to zero. In

[5] For the interested reader, we want to state that the reason here seems to be that the subroutine has to guess more than one image of the linear transformations that map S to its linear representative. Intuitively, it is clear that we need more guesses if there are self-equivalences.

fact, since we rely on *affine* equivalence, the value of those bits does not matter. The two identified control inputs do not influence the computation of the S-box; hence, we do not have to consider them here.

Once the Boolean functions have been extracted, we run Algorithm 2. Crucially, our S-box database contains the AES S-box. Consequently, our implementation of Algorithm 2 correctly identifies the extracted AES S-box as such in a fraction of a second on a consumer-grade laptop. As the details of the `linear_representation` algorithm and our implementation of it are rather unrelated to the topic of this work, we do not go into the details but refer the interested reader to [15] and our source code.[6]

Extracting the Linear Layer. If we only want to know whether our candidate implements a cryptographic algorithm, we can stop as soon as we identify the first S-box. Still, if we want to restore the linear layer, we need to identify all of its input and output gates again. Hence, in the case of $\mathcal{L} \circ \mathcal{S}$, the first step of extracting the linear layer is actually to restore *all* the S-boxes because the outputs of the S-boxes are the inputs to the linear layer. Obviously, recovering *all* S-boxes might be harder than recovering only a single one. For instance, consider SKINNY which features a structure similar to AES, but the 4×4 `MixColumn` matrix is binary, i.e., no finite field multiplications take place. Therefore, the outputs of its lightweight S-boxes are directly xored with each other, which can lead to circuit optimizations that make it hard to extract the SKINNY S-boxes from the netlist. However, in SKINNY the first row of the state is just moved to the second row. Here, the four corresponding S-boxes are easier to extract than all the others.

If the extraction of all S-boxes succeeds, extracting the linear function between the outputs of all S-boxes and the output register is straightforward. We simply evaluate the Boolean functions of the linear layer on all unit vectors and use the results to build a matrix. If the function is affine and not linear, we simply evaluate the all-zero input to learn the constant. To double-check that the function is indeed linear (or affine), we can evaluate it on random inputs and compare the results to those that we get using the recovered matrix (and constant). Of course, the same caveats about additional inputs apply again.

There is, however, another caveat, namely the ordering of the inputs and outputs. Given that we do not know the order of the flip-flops within our recovered state registers, they are equal to the actual state registers only up to a permutation of the bits. Consider the block cipher PRESENT, for which the linear layer is just a bit permutation. In other words, bit permutations can be sufficient to build linear layers and therefore a truly solid linear layer extraction would need to recover the exact linear layer and not only the linear layer up to a permutation of the input and outputs. Of course, if we detect a *known* S-box, we could try to find the linear layer of the matching cipher. But as soon as the S-box is unknown, this is not possible anymore and hence we believe that the most reasonable approach then is to fall back to a semi-automated approach.

[6] See Footnote 4.

Example 5 (Round-based AES). Our running example illustrates one of the inconveniences we have to deal with: it features control inputs that change the functionality of the linear layer. However, there are only two control inputs; hence, we can simply restore the linear layer for all four possible control states. If we do so, we get a zero matrix twice, a permutation matrix, and then a more complex matrix M. We consider the linear layer for each component separately, i.e., all matrices are 32×32 binary matrices. As discussed before, those matrices represent the linear layer only up to a permutation of the inputs and outputs. Since we already know that we are examining the AES by identifying its S-boxes, reordering the inputs and outputs is straightforward. If we do so, the permutation matrix becomes the identity and M becomes the `MixColumn` matrix. Against this background, it becomes apparent that the two control inputs determine whether the plaintext (additional input set to zero) is loaded and whether `MixColumns` takes place.

3.4 Analyzing ARX and Shift Registers

Certainly, SPNs are not the only strategy to design modern symmetric ciphers. Two other approaches are ARX constructions and shift register-based ciphers. Well-known examples for ARX, i.e., ciphers based on modular **a**ddition, **r**otation and **X**OR, are ChaCha and SHA-2. Arguably, a cipher like ChaCha was built to be efficient in software and hence searching for it in hardware is only of low interest. SHA-2, however, which is implemented in innumerable Bitcoin mining accelerators, surely demonstrates that ARX primitives are also implemented in hardware. Admittedly, we did not investigate ARX in detail. Still, since the overall structure for such ARX ciphers is also based on iterated rounds, we can actually make use of the modularity of our tool and use prior work on detecting additions in hardware [40,53] to check whether detected round functions incorporate additions. We believe this to be a good indicator for ARX ciphers.

In recent years, symmetric cryptography based on shift registers got a lot of attention in the context of backdoors. That is, it was discovered that the secret cipher suites GEA [9] and TEA [46], both standardized by the European Telecommunications Standards Institute (ETSI), contain deliberate weaknesses. As already mentioned in the introduction, uncovering such proprietary or secret ciphers was a clear motivation for developing `HAWKEYE`. Although we did not aim to detect shift-register-based ciphers, depending on the specific implementation, `HAWKEYE` can already detect some of them in gate-level netlists, see Section 4.4. The reason for this is that partially unrolled implementations of, e.g., Trivium look similar to implementations of block ciphers.

4 Evaluation

4.1 Methodology

To evaluate our approach, we implemented `HAWKEYE` as a plugin to the open-source netlist analysis framework HAL [33,37]. The core of the plugin is written

in C++, but can be interacted with using Python. This plugin is available in the HAL GitHub repository.[7] All benchmark netlists discussed in the following will be made available in the artifacts associated with this paper.[8] The runtimes given in this section were derived using an Apple Macbook Pro with an M2 Max processor.

Access to real-world devices containing secret or proprietary ciphers is limited and the engineering overhead of netlist extraction from them would far exceed the scope of this work, cf. Section 2.2. However, gate-level netlists are not only encountered during reverse engineering, but also as part of the hardware design flow. Hence, we can simply synthesize ciphers on our own to generate the respective gate-level netlists and then use them to evaluate our approach. To generate these benchmarks, we searched for public hardware implementations of (primarily) symmetric ciphers. We tested on third-party designs to reduce the bias of trying to detect something that we ourselves created. We leveraged Xilinx Vivado to synthesize the benchmarks for Xilinx 7-Series FPGAs and Synopsys Design Compiler with the Synopsys LSI_10k technology library to generate ASIC benchmarks. We chose these tools since Xilinx is the world's largest FPGA vendor and Synopsys dominates the electronic design automation tool market. This way, we demonstrate that our approach is technology agnostic and even works across platforms. The aforementioned design tools retain meaningful labels in the gate-level netlists after synthesis that allow us to manually identify state registers based on flip-flop names and thereby generate a ground truth for evaluation. Of course, `HAWKEYE` does not interpret these labels during analysis.

Based on our findings, we discuss key differences between FPGA and ASIC implementations, the challenges that arise therefrom, and how to address them. In Section 4.2, we demonstrate the effectiveness of our approach on OpenTitan, an industry-grade open-source hardware design of a root of trust security SoC. Thereby, we show that `HAWKEYE` is effective in detecting implementations of symmetric ciphers even in large gate-level netlists comprising many components other than cryptography. Based on our findings on a smaller SoC in Section 4.3, we argue that cryptographic and non-cryptographic components of a larger design can be analyzed separately when evaluating our approach. This allows us to run `HAWKEYE` on a vast number of small benchmark designs in Section 4.4.

4.2 OpenTitan – Case Study on a Secure Micro-Controller

OpenTitan is «the first open source project building a transparent, high-quality reference design and integration guidelines for silicon root of trust chips» [43]. Within this ecosystem, Earl Grey is a standalone secure micro-controller design that implements a RISC-V CPU, a big-number accelerator, cryptographic accelerators, interfaces, and some periphery, see Fig. 6. Earl Grey is one of only a few industry-grade open-source hardware implementations. To the best of our

[7] See Footnote 4.
[8] See https://artifacts.iacr.org/.

knowledge, it is the only such design of a secure microcontroller featuring implementations of cryptographic algorithms. Namely, it implements AES with 128, 192, and 256-bit key sizes, an HMAC based on SHA-256, a KMAC, a Xoshiro256++ PRNG, and memory scramblers based on round-reduced variants of PRINCE and PRESENT. The OpenTitan project is administered by the lowRISC initiative [42] and supported by industry partners such as Google.

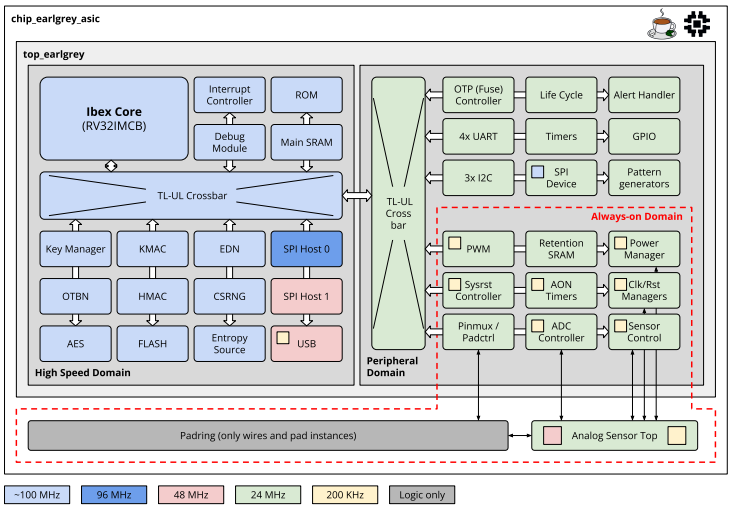


Fig. 6: Overview of the Earl Grey hardware architecture taken from [44].

We evaluated HAWKEYE on a self-synthesized FPGA implementation of Earl Grey. While Earl Grey offers on-demand side-channel protections that can be configured during design, we did not enable them as our work does not deal with side-channel-protected cipher implementations. The resulting gate-level netlist contains 424 341 gates. Running the state register identification (see Section 3.1) of HAWKEYE takes 44.3 seconds when using Method 1, i.e., filtering flip-flops by their control inputs. This produces the 25 candidates listed in Table 2. When additionally using Method 2, i.e., SCC detection, the runtime slightly increases to 46.6 seconds, and 21 candidates are found (Candidates 1-21). Obviously, HAWKEYE also identifies candidates not related to cryptographic implementations. Still, at 12 out of 25 (or 21) candidates actually implementing cryptography primitives, it provides surprisingly accurate results in a matter of seconds, a task that would otherwise take a human weeks if not months to complete. Employing SCC-based detection reduces the number of false positives from 13 to 9 while detecting the same cryptographic implementations as before, thereby lowering the manual verification effort even further. Hence, we observe SCC detection generally improving our results, at the cost of a higher runtime.

True Positives. Of the 1600-bit Keccak state register, `HAWKEYE` identifies 640 flip-flops. This is the case because some flip-flops of this register rely on different enable signals. Given that the number of 640 flip-flops directly relates to the message pipeline size of 10×64 bits, we assume that the synthesizer somehow takes the state of the message pipeline into account when enabling 640 of the Keccak state flip-flops. The state flip-flops of the AES round function, the AES key schedule, SHA-2, and Xoshiro256++ are accurately found. For the candidate that is the correct state register of the AES round function, `HAWKEYE` automatically extracts and identifies its S-box. For PRNGs such as Xoshiro256++, we consider them to be a true positive despite them not being cryptographically secure because a priori it cannot be known whether such LFSRs are used in a cryptographic setting. Both PRESENT and PRINCE are used for memory scrambling and are implemented in round-reduced, unrolled variants to boost performance. As `HAWKEYE` is not designed to detect unrolled implementations, we assume that these candidates have been found because of surrounding registers that have a size equal to the respective candidate's output register. In addition to all these cryptographic primitives, we find five PRNGs that are mostly used for wiping secret keys or data in general. Many of these PRNGs are implemented using LFSRs, detection of which was not among `HAWKEYE`'s initial design goals. Still, these results hint at our approach also being (at least to some extent) useful to find LFSR-based implementations.

We note that we cannot generate a fully accurate ground truth for Earl Grey, as some of the primitives discovered during our evaluation are not even mentioned in the official documentation. Exhaustive manual analysis of the entire netlist to generate such a ground truth was also not feasible. Still, we know that Earl Grey features additional PRINCE implementations at various memory interfaces, which we did not detect due to them being fully unrolled.

False Positives. Two false-positive candidates identified by `HAWKEYE` (Candidates 13 and 14) can be attributed to error-correcting codes that are sometimes implemented using structural properties similar to those of symmetric cryptography. Of these two, Candidate 14 additionally implements a multiply-accumulate unit where the accumulator also forms a round-based structure. In addition, the four discovered 40-bit counters are used to keep track of the LFSRs of Candidates 11 and 12 in order to reset them once their maximum output length has been reached. Counters feature a register for the current counter state and combinational logic to update the state based on the current counter value, which is similar to a round-based cipher implementation. Another class of false-positives can be attributed to FSMs that usually exhibit a structure similar to counters. Particularly in the security-related components of Earl Grey, these are implemented using sparse, one-hot encoding. This encoding choice results in large state register sizes, especially when counters are implemented as part of such an FSM. `HAWKEYE` also finds two candidates that are the input packers for the HMAC and KMAC implementations, respectively. Although not completely unrelated to cryptography, we view them as utility circuitry and classify them as

false positives. Finally, we note that when additionally using Method 2, we get refined results in that the false-positive candidates 22 to 25 are no longer found.

Table 2: Overview of the candidates that HAWKEYE found for OpenTitan's Earl Grey. For every candidate, we list the number of flip-flops it contains, indicate whether it is indeed a cryptographic primitive, and provide a brief description.

No.	#FFs	Crypto?	Description
1	640	✓	partial Keccak state
2	128	✓	AES state
3	256	✓	AES round key
4	256	✓	SHA-2 state
5	256	✓	Xoshiro256++ state
6	192	✓	PRESENT state and key
7	64	✓	PRINCE output
8	64	✓	LFSR of PRNG within analog sensors
9	64	✓	key manager clearing PRNG
10	64	✓	AES clearing PRNG
11	40	✓	LFSR of PRNG in memory controller
12	40	✓	LFSR of PRNG in memory controller
13	1153	✗	error-correcting code output
14	312	✗	bignum multiply-accumulate with error-correcting code
15	40	✗	counter in memory controller
16	40	✗	counter in memory controller
17	40	✗	counter in memory controller
18	40	✗	counter in memory controller
19	384	✗	secured state machine
20	320	✗	secured state machine
21	96	✗	secured state machine
22	44	(✗)	SPI Host 1 state machine
23	44	(✗)	SPI Host 0 state machine
24	135	(✗)	HMAC packer
25	88	(✗)	KMAC packer

4.3 Cryptographic Accelerators in a Small SoC

Given the complexity of OpenTitan, we conduct additional experiments on a smaller SoC that allows us to easily swap the implemented cryptographic primitives. To this end, we chose the WBSoC benchmark provided by the *CAD For Assurance* [22] project. In our configuration, it contains a PicoRV32 RISC-V CPU, some memory, a UART interface, and a 32-bit Wishbone bus connecting the individual components. In addition, we adapted the SoC design such that a single cryptographic primitive is implemented at a time. For this purpose, we

synthesized the SoC once for each of three different cryptographic algorithms, namely AES-128, ASCON [30], and SHA-256. As the SoC has been designed for Xilinx FPGAs, we could not synthesize it for ASICs.

Table 3: Overview of our results for the three different WBSoC implementations. We deem an experiment successful if one of our methods finds a cryptographic state register or if no false-positive candidates are discovered for noise benchmarks. TP and FP are the number of true and false positives, respectively. Runtimes are given in columns t. ***S*?** denotes whether the S-box of the cipher was identified.

Name	**Suc.**	**Method 1**			**Method 2**			***S*?**	**Gates**	**Src.**
		TP	*FP*	*t*	*TP*	*FP*	*t*			
AES-128	✓	1	4	0.08s	2	1	1.13s	✓	4830	[1]
ASCON	✓	0	4	0.18s	1	1	1.76s	✓	4962	[60]
SHA-256	✓	1	4	0.27s	1	1	2.77s	-	5655	[58]

We analyzed all three resulting gate-level netlists with `HAWKEYE`, the results of these experiments are given in Table 3. To this end, we initially set $s_{\min} = 32$ and tested two different `HAWKEYE` configurations. First, we tested our candidate search using Method 1, hence filtering by flip-flop types and control inputs before assembling the neighborhoods. Second, we executed this search again while only using Method 2, i.e., applying SCC detection on the neighborhoods.

`HAWKEYE` detects correct candidates for AES-128 and SHA-256 even without additional SCC detection, although the number of false positives declines when resorting to SCC detection instead. ASCON is not detected this way, because in our implementation some flip-flops of the state are enabled by different inputs than others. Hence, `HAWKEYE` can only locate ASCON with SCC detection enabled and flip-flop filtering disabled. The increased runtime of `HAWKEYE` with SCC detection can mostly be attributed to the missing control input checks, as this causes more flip-flops to be considered during analysis. We observe that the runtime increases with the state size of the implemented algorithm. AES-128 features a 128-bit state, ASCON a 320-bit state, and SHA-2 a 256-bit state.

Regarding the false positives during candidate search, we find that `HAWKEYE` always detects the very same candidates because the logic surrounding each of the three cryptographic implementations does not change. When searching without SCC detection, two false candidates of size 33 and two of size 32 are detected independent of the implemented cipher. Hence, this indicates that increasing $s_{\min}$ to, e.g., 40 helps reduce false positives without affecting the identification of correct candidates. Similarly, we determined that setting $c_{\min} = 10$ reduces the number of false positives even further compared to other values of $c_{\min}$.

4.4 Isolated (Non-)Cryptographic Benchmarks

Based on our findings in Section 4.3, we argue that finding a cryptographic implementation with HAWKEYE is independent of its surroundings in that the implementation of, e.g., the PicoRV32 CPU does not influence the detection of cryptographic candidates. Hence, we evaluated HAWKEYE on a wide selection of standalone open-source implementations of cryptographic primitives and "noise", i.e., implementations of non-cryptographic hardware designs such as CPUs and signal processing modules. To show the applicability of HAWKEYE beyond FPGAs netlists, we synthesized all benchmark netlists for both ASICs and FPGAs as described in Section 4.1. We refer to Table 4 for our evaluation results on FPGA and ASIC netlists. For this part of our evaluation, we only consider a candidate a true positive if it contains the full state and/or round-key register. Partial hits are considered a false positive and are discussed in more detail below. We do not expect many other false positives as the benchmarks implement nothing but the cipher and an additional UART interface for communication.

FPGA Benchmarks. HAWKEYE successfully locates the three DES state registers of 3DES, even without additional SCC detection. In general, we observe that SCC detection is rarely needed to locate cryptographic implementations on FPGA netlists. As we have seen in the two previous case studies, it can still help reduce false positives. However, for pipelined implementations such as the AES-128$_p$ and DES benchmarks, enabling SCC detection will actually prevent HAWKEYE from locating these primitives, as pipelined implementations do not form an SCC in the first place. For AES-128$_p$, we do not find any candidates as its S-boxes are implemented in memory, which we did not consider when designing our algorithms. Extending HAWKEYE to deal with this issue is straightforward, but we did not implement this to avoid over-fitting to our benchmarks. For the pipelined DES, a Feistel cipher with 16 rounds, we find 13 state registers. This is expected, as three iterations of our candidate search algorithm must be executed before the state size does not grow anymore and the algorithm terminates.

AES-128$_r$ is a round-based AES implementation, for which HAWKEYE swiftly finds the state register when applying flip-flop filtering and the state register and round-key register when instead resorting to SCC detection. A similar behavior can be observed for PRESENT-80 and PRESENT-128 [18]. Furthermore, when using SCC detection we not only find the main 256-bit state of SHA-256, but also another 256-bit intermediate register, resulting in a single 512-bit candidate. For SIMON, we only find the round-key register and not the state register when using Method 2, most likely because HAWKEYE times out before the rather slow diffusion of SIMON causes the state size to saturate if SCCs are considered. For many other ciphers, there is no difference in the number of results found when using Method 2 compared to when Method 1 is used.

In addition to SPN, Feistel, and ARX ciphers, we also tested HAWKEYE on benchmarks that it was not designed to handle. For the NLFSR-based KATAN and KTANTAN [23] ciphers, it finds the key register and state register respectively only when SCC detection is enabled and control input checks are disabled.

It does not find the state register for KATAN, but only its key register. Surprisingly, for KTANTAN it only finds the state register, but not the key register. PRINCE is implemented fully unrolled, i.e., without any registers in between the individual rounds. `HAWKEYE` is not designed to detect such implementations, which is why no candidate is identified. When running `HAWKEYE` on RSA-512, we find candidates that correspond to the registers of the multiplication and square units of the RSA implementation, but we do not consider them to be a hit, despite them most likely being useful for further manual analysis.

We additionally experimented with the NLFSR-based cipher Trivium [24], as it is designed especially with hardware implementations in mind. It allows parallel computation of multiple NLFSR bits at once, which results in a structure similar to a round-based cipher. Hence, we set out to explore whether `HAWKEYE` can detect such implementations and how much parallelization is required for reliable detection. Using Method 1, we find parts of the state register only for Trivium_{64}, i.e., an implementation where 64 bits of the next NLFSR state are computed in parallel. When using Method 2, we can reliably detect the state register starting at 32 bits of parallelization. Together with our observations on KATAN and KTANTAN as well as in Section 4.2, this unintended finding hints at `HAWKEYE` being partially applicable to ciphers based on shift registers on FPGAs as well.

Regarding the S-box analysis, we find that `HAWKEYE` successfully extracts and identifies ciphers with round functions essentially consisting only of a full S-box layer and a linear layer. Exceptions are SHA-3 and Piccolo [63] because of their dense linear layer in combination with lightweight S-boxes. For (3)DES [54] and Magma [31], the expansion step and the *modular* key addition respectively, each taking place before the S-box layer, introduce dependencies that prevent successful S-box detection. As we found no candidate for PRINCE, we cannot recover an S-box.

ASIC Benchmarks. In ASIC netlists, we observe that many of the synchronous control inputs that were directly connected to flip-flops in FPGA netlists and helped us to differentiate the state from the ciphertext register are now implemented in the combinational fan-in of the flip-flop data inputs. Hence, for round-based implementations, we cannot properly prevent `HAWKEYE` from escaping the state computation during candidate search, which is also reflected in our low detection rates when using Method 1. To resolve this issue, we initially experimented with identifying and tracing control inputs through the combinational logic by computing intersections of inputs to the combinational fan-in of connected flip-flops, but this did not produce reliable results and came with a hefty runtime overhead. In the end, we opted to follow our SCC-based approach known as Method 2, as it performed better across all of our benchmarks.

For pipelined implementations such as AES-128_p and DES, this issue does not persist since one state register is always only followed by another state register but has no additional outputs. Because AES-128_p is fully implemented in combinational logic for ASICs—in contrast to FPGAs, where the S-boxes were

implemented in memory—, HAWKEYE now also reliably detects 7 of its state registers. Using Method 2 prevents HAWKEYE from finding pipelined implementations, as previously observed for the FPGA benchmarks.

For most ASIC benchmarks, HAWKEYE produced results similar to the FPGA benchmarks when using Method 2. Most notably, it only finds two out of three state registers for 3DES in the ASIC case, which is still sufficient for further analysis. For PRINCE, HAWKEYE finds 193 partially overlapping false positive candidates. Surprisingly, for ASIC netlists, HAWKEYE is better in locating Trivium with less parallelization, which is a contradictory finding to the FPGA case.

Generally, longer runtimes can be observed for ASIC netlists compared to FPGA netlists, as they require more combinational gates to implement the same functionality as LUTs in FPGAs. This also increases the size of the graph that HAWKEYE needs to traverse to identify candidates. Regarding S-box detection, we get slightly worse results compared to FPGAs. More specifically, we can no longer identify the S-boxes of ASCON and Midori [6]. Furthermore, HAWKEYE correctly isolates the S-boxes of GIFT [7] and PRESENT-128 but cannot restore them due to some misclassified control inputs.

Noise Benchmarks. For both ASICs and FPGAs, we observe at most three false positives when running HAWKEYE on non-cryptographic benchmarks. For most of them, we actually only find no or one incorrect candidate. Upon closer inspection, almost all identified candidates exhibit sizes that are atypical for cryptographic implementations. Hence, we argue that such a low number of false positives combined with straightforward manual verification can be tolerated.

Table 4: Our results for FPGA and ASIC. See Table 3 for explanation of the columns.

Name	FPGA								ASIC								Src.
	Suc.	Method 1			Method 2			S?	Gates	Suc.	Method 1			Method 2			S?	Gates	
		TP	*FP*	*t*	*TP*	*FP*	*t*				*TP*	*FP*	*t*	*TP*	*FP*	*t*			
3DES	✓	3	0	0.05s	3	0	0.30s	✗	3 458	✓	0	1	0.25s	2	0	0.08s	✗	9 916	[58]
AES-128$_r$	✓	1	0	0.06s	2	0	0.40s	✓	3 458	✓	0	1	0.79s	2	0	0.53s	✓	12 804	[1]
AES-128$_p$	✗	0	0	0.02s	0	0	0.02s	✗	8 064	✓	7	0	0.62s	0	0	1.98s	✗	154 477	[2]
ASCON	✓	0	0	0.11s	1	0	1.64s	✓	5 323	✓	0	1	2.44s	1	0	2.21s	✗	11 786	[60]
CRAFT	✓	1	0	0.01s	1	0	0.22s	✓	1 547	✓	0	1	0.25s	1	0	0.26s	✓	3 503	[1]
DES	✓	13	0	0.09s	0	0	0.14s	✗	3 956	✓	13	0	0.15s	0	0	0.20s	✗	19 976	[2]
GIFT	✓	1	0	0.01s	1	0	0.08s	✓	1 512	✓	0	1	0.15s	1	0	0.12s	(✓)	2 969	[1]
LED-64	✓	1	0	0.01s	1	0	0.06s	✓	1 328	✓	0	1	0.20s	1	0	0.09s	✓	2 798	[1]
LED-128	✓	1	0	0.01s	1	0	0.09s	✓	1 541	✓	0	1	0.30s	1	0	0.14s	✓	2 989	[1]
Magma	✓	1	0	0.02s	1	0	0.17s	✗	1 808	✓	1	0	0.56s	1	0	0.13s	✗	4 894	[58]
Midori	✓	1	0	0.01s	1	0	0.11s	✓	1 489	✓	0	1	0.28s	1	0	0.17s	✗	2 824	[1]
Piccolo	✓	1	0	0.01s	1	0	0.10s	✗	1 405	✓	0	1	0.27s	1	0	0.14s	✗	3 500	[1]
PRESENT-80	✓	1	0	0.02s	2	0	0.02s	✓	1 302	✓	1	0	0.05s	2	0	0.04s	✓	2 937	[58]
PRESENT-128	✓	1	0	0.02s	2	0	0.08s	✓	1 538	✓	0	1	0.17s	2	0	0.14s	(✓)	3 486	[1]
SHA-256	✓	1	0	0.27s	1	0	1.64s	-	3 547	✓	0	2	0.77s	1	0	2.94s	-	7 152	[58]
SHA-3	✓	1	0	5.50s	1	0	16.11s	✗	8 143	✓	0	1	9.65s	1	0	15.87s	✗	22 414	[58]
SIMON-128	✓	1	0	0.01s	1	0	0.11s	-	1 539	✓	0	1	0.09s	1	0	0.16s	-	3 620	[1]
SKINNY-64	✓	1	0	0.01s	1	0	0.07s	✓	1 221	✓	0	1	0.11s	1	0	0.11s	✓	2 197	[1]

Table 4: (continued).

Name	FPGA								ASIC								Src.
	Suc.	Method 1			Method 2			*S*?	Gates	Suc.	Method 1			Method 2			*S*?	Gates	
		TP	*FP*	*t*	*TP*	*FP*	*t*				*TP*	*FP*	*t*	*TP*	*FP*	*t*			
KATAN-80	✓	0	0	0.01s	1	0	0.03s	-	1 246	✓	0	1	0.06s	1	0	0.07s	-	3 160	[1]
KTANTAN-80	✓	0	0	0.00s	1	0	0.03s	-	1 118	✓	0	1	0.06s	1	0	0.08s	-	2 543	[1]
PRINCE	✗	0	1	0.06s	0	0	0.15s	✗	2 600	✗	0	193	0.34s	0	0	0.42s	✗	7 238	[39]
RSA-512	✗	0	4	8.55s	0	2	9.96s	-	20 908	✗	0	3	20.56s	0	3	14.35s	-	82 167	[2]
Trivium_8	✗	0	0	0.01s	0	1	0.01s	-	1 225	✓	0	0	0.05s	1	0	0.05s	-	3 022	[25]
Trivium_{32}	✓	0	0	0.01s	1	0	0.09s	-	1 427	✗	0	1	0.08s	0	1	0.19s	-	2 867	[25]
Trivium_{64}	✓	0	1	0.02s	1	0	0.49s	-	1 683	✗	0	1	0.19s	0	1	0.35s	-	3 532	[25]
Ethernet	✓	-	0	0.05s	-	0	0.27s	-	5 676	✗	-	1	19.83s	-	1	0.47s	-	42 161	[58]
CIC filter	✓	-	0	0.01s	-	0	0.01s	-	572	✓	-	0	0.02s	-	0	0.05s	-	1 221	[58]
Hilbert trans.	✓	-	0	0.02s	-	0	0.03s	-	1 390	✓	-	1	0.07s	-	0	0.06s	-	3 004	[58]
CPU Edge	✗	-	0	0.61s	-	1	2.23s	-	11 144	✗	-	3	2.71s	-	3	2.86s	-	41 909	[2]
CPU Ibex	✗	-	1	0.43s	-	1	1.21s	-	6 340	✗	-	1	20.06s	-	1	36.23s	-	12 751	[2]
CPU open8	✗	-	0	0.02s	-	1	0.02s	-	1 216	✗	-	0	0.02s	-	1	0.02s	-	1 888	[2]

5 Conclusion

In this work, we presented the first comprehensive approach dedicated to the recovery of implementations of symmetric cryptography from netlists, i.e., gate-level descriptions of hardware. Overall, our evaluation proved our techniques to be instrumental and efficient. Generally, despite one failing FPGA benchmark, reliability, runtime, and accuracy of `HAWKEYE` are better for FPGA netlists than for ASIC ones. Still, `HAWKEYE` generates convincing results in both cases. Of the two methods we proposed for candidate detection, Method 1 performs best for FPGA designs, while Method 2 is better suited for ASIC netlists. However, to check for pipelined implementations on ASICs, Method 1 should always be executed on them as well. Automated S-box extraction not only proved helpful to identify known cryptographic algorithms but could also aid in pin-pointing proprietary ciphers.

Limitations and Future Work. Certainly, there are interesting challenges left that have not been addressed by our work. For instance, we saw that `HAWKEYE` produces some typical false positives, especially for state machines, counters, and error-correcting codes. Additional filters should be developed in the future to refine results. Moreover, the extraction of S-boxes and linear layers can be improved. In particular, `HAWKEYE` cannot restore S-boxes from memory and struggles with lightweight S-boxes in combination with a dense linear layer or if there are complex operations in front of the S-box. Incorporating the first is straightforward, but for the latter, improved heuristics are needed. The extraction of the linear layer is only semi-automated. For full automation, it would be necessary to study consecutive rounds instead of only a single isolated round function. Furthermore, we mostly ignored unrolled implementations, feedback shift registers, and (side-channel) countermeasures. Those surely are interesting topics for future work, as is applying `HAWKEYE` to proprietary real-world implementations.

Another line of future work is to study countermeasures against `HAWKEYE` and how to overcome such protections again. That is, a designer aware of our approach could defeat `HAWKEYE`. Similarly, once an attacker becomes aware of additional protections, it is likely easy to adapt `HAWKEYE` to beat them. Hence, we expect this to become a cat-and-mouse game. Putting S-boxes in memory and using side-channel protections will significantly hamper `HAWKEYE` for now, but only until it is adapted to tackle these challenges. Finally, we believe some of our techniques could potentially be transferred to software reverse engineering.

Acknowledgments. We thank Armand Schinkel for his assistance in developing the first proofs-of-concept for `HAWKEYE` and Thorben Moos for synthesizing the ASIC benchmarks. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany´s Excellence Strategy - EXC 2092 CASA – 390781972. This work has been funded in parts by the ERC project 101097056 (SYMTRUST).

References

1. Aghaie, A., Moradi, A., Rasoolzadeh, S., Shahmirzadi, A.R., Schellenberg, F., Schneider, T.: Impeccable circuits. IEEE Trans. Computers **69**(3), 361–376 (2020). https://doi.org/10.1109/TC.2019.2948617
2. Albartus, N., Hoffmann, M., Temme, S., Azriel, L., Paar, C.: DANA universal dataflow analysis for gate-level netlist reverse engineering. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(4), 309–336 (2020). https://doi.org/10.13154/tches.v2020.i4.309-336
3. Alliance, C.: Project X-Ray, https://github.com/f4pga/prjxray
4. Appelbaum, J.: Communication in a world of pervasive surveillance: Sources and methods: Counter-strategies against pervasive surveillance architecture. Phd thesis 1 (research tu/e / graduation tu/e), Mathematics and Computer Science (Mar 2022), proefschrift.
5. Azriel, L., Speith, J., Albartus, N., Ginosar, R., Mendelson, A., Paar, C.: A survey of algorithmic methods in IC reverse engineering. J. Cryptogr. Eng. **11**(3), 299–315 (2021). https://doi.org/10.1007/s13389-021-00268-5
6. Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A block cipher for low energy. In: Iwata, T., Cheon, J.H. (eds.) Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9453, pp. 411–436. Springer (2015). https://doi.org/10.1007/978-3-662-48800-3_17
7. Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT: A small present - towards reaching the limit of lightweight encryption. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10529, pp. 321–345. Springer (2017). https://doi.org/10.1007/978-3-319-66787-4_16
8. Baudrin, J., Boeuf, A., Couvreur, A., Joly, M., Perrin, L.: SboxU (2023), https://github.com/lpp-crypto/sboxU/
9. Beierle, C., Derbez, P., Leander, G., Leurent, G., Raddum, H., Rotella, Y., Rupprecht, D., Stennes, L.: Cryptanalysis of the GPRS encryption algorithms GEA-1 and GEA-2. In: Canteaut, A., Standaert, F. (eds.) Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12697, pp. 155–183. Springer (2021). https://doi.org/10.1007/978-3-030-77886-6_6
10. Beierle, C., Felke, P., Leander, G., Rønjom, S.: Decomposing linear layers. IACR Trans. Symmetric Cryptol. **2022**(4), 243–265 (2022). https://doi.org/10.46586/tosc.v2022.i4.243-265
11. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY family of block ciphers and its low-latency variant MANTIS. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9815, pp. 123–153. Springer (2016). https://doi.org/10.1007/978-3-662-53008-5_5

12. Benz, F., Seffrin, A., Huss, S.A.: Bil: A tool-chain for bitstream reverse-engineering. In: Koch, D., Singh, S., Tørresen, J. (eds.) 22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012. pp. 735–738. IEEE (2012). https://doi.org/10.1109/FPL.2012.6339165
13. Bernstein, D.J.: Chacha, a variant of salsa20 (2008), https://cr.yp.to/chacha/chacha-20080128.pdf
14. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7881, pp. 313–314. Springer (2013). https://doi.org/10.1007/978-3-642-38348-9_19
15. Biryukov, A., Cannière, C.D., Braeken, A., Preneel, B.: A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In: Biham, E. (ed.) Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2656, pp. 33–50. Springer (2003). https://doi.org/10.1007/3-540-39200-9_3
16. Biryukov, A., Perrin, L., Udovenko, A.: Reverse-engineering the s-box of streebog, kuznyechik and stribobr1. In: Fischlin, M., Coron, J. (eds.) Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9665, pp. 372–402. Springer (2016). https://doi.org/10.1007/978-3-662-49890-3_15
17. Biryukov, A., Shamir, A.: Structural cryptanalysis of SASAS. J. Cryptol. **23**(4), 505–518 (2010). https://doi.org/10.1007/s00145-010-9062-1
18. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer (2007). https://doi.org/10.1007/978-3-540-74735-2_31
19. Bono, S., Green, M., Stubblefield, A., Juels, A., Rubin, A.D., Szydlo, M.: Security analysis of a cryptographically-enabled RFID device. In: McDaniel, P.D. (ed.) Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005. USENIX Association (2005), https://www.usenix.org/conference/14th-usenix-security-symposium/security-analysis-cryptographically-enabled-rfid-device
20. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.: PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In: Wang, X., Sako, K. (eds.) Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7658, pp. 208–225. Springer (2012). https://doi.org/10.1007/978-3-642-34961-4_14
21. Brunner, M., Baehr, J., Sigl, G.: Improving on state register identification in sequential hardware reverse engineering. In: IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019. pp. 151–160. IEEE (2019). https://doi.org/10.1109/HST.2019.8740844

22. CADForAssurance: System on chip benchmarks (2020), https://cadforassurance.org/soc-platform/soc-benign-benchmark/system-on-chip-benchmarks/
23. Cannière, C.D., Dunkelman, O., Knezevic, M.: KATAN and KTANTAN - A family of small and efficient hardware-oriented block ciphers. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings. Lecture Notes in Computer Science, vol. 5747, pp. 272–288. Springer (2009). https://doi.org/10.1007/978-3-642-04138-9_20
24. Cannière, C.D., Preneel, B.: Trivium. In: Robshaw, M.J.B., Billet, O. (eds.) New Stream Cipher Designs - The eSTREAM Finalists, Lecture Notes in Computer Science, vol. 4986, pp. 244–266. Springer (2008). https://doi.org/10.1007/978-3-540-68351-3_18
25. Cassiers, G., Masure, L., Momin, C., Moos, T., Moradi, A., Standaert, F.: Randomness generation for secure hardware masking - unrolled trivium to the rescue. IACR Cryptol. ePrint Arch. p. 1134 (2023), https://eprint.iacr.org/2023/1134
26. Daemen, J., Rijmen, V.: The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition. Information Security and Cryptography, Springer (2020). https://doi.org/10.1007/978-3-662-60769-5
27. Defense Express Media & Consulting Company: Encryption microchip from aliexpress found inside russian portable "azart" transceivers (2022), https://en.defence-ua.com/weapon_and_tech/encryption_microchip_from_aliexpress_found_inside_russian_portable_azart_transceivers-4907.html
28. Ding, Z., Wu, Q., Zhang, Y., Zhu, L.: Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. Microprocess. Microsystems **37**(3), 299–312 (2013). https://doi.org/10.1016/j.micpro.2012.12.003
29. Dinur, I.: An improved affine equivalence algorithm for random permutations. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I. Lecture Notes in Computer Science, vol. 10820, pp. 413–442. Springer (2018). https://doi.org/10.1007/978-3-319-78381-9_16
30. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2: Lightweight authenticated encryption and hashing. J. Cryptol. **34**(3), 33 (2021). https://doi.org/10.1007/s00145-021-09398-9
31. Dolmatov, V., Baryshkov, D.: Gost r 34.12-2015: Block cipher "magma". Tech. Rep. RFC 8891 (September 2020), https://www.rfc-editor.org/rfc/rfc8891
32. Eisenbarth, T., Kasper, T., Moradi, A., Paar, C., Salmasizadeh, M., Shalmani, M.T.M.: On the power of power analysis in the real world: A complete break of the keeloqcode hopping scheme. In: Wagner, D.A. (ed.) Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5157, pp. 203–220. Springer (2008). https://doi.org/10.1007/978-3-540-85174-5_12
33. Embedded Security Group: HAL - The Hardware Analyzer. https://github.com/emsec/hal (2019)
34. Ender, M., Leander, G., Moradi, A., Paar, C.: A cautionary note on protecting xilinx' ultrascale(+) bitstream encryption and authentication engine. In: 30th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2022, New York City, NY, USA, May 15-18, 2022. pp. 1–9. IEEE (2022). https://doi.org/10.1109/FCCM53951.2022.9786118

35. Ender, M., Moradi, A., Paar, C.: The unpatchable silicon: A full break of the bitstream encryption of xilinx 7-series fpgas. In: Capkun, S., Roesner, F. (eds.) 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020. pp. 1803–1819. USENIX Association (2020), https://www.usenix.org/conference/usenixsecurity20/presentation/ender
36. Ender, M., Swierczynski, P., Wallat, S., Wilhelm, M., Knopp, P.M., Paar, C.: Insights into the mind of a trojan designer: the challenge to integrate a trojan into the bitstream. In: Shibuya, T. (ed.) Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019. pp. 112–119. ACM (2019). https://doi.org/10.1145/3287624.3288742
37. Fyrbiak, M., Wallat, S., Swierczynski, P., Hoffmann, M., Hoppach, S., Wilhelm, M., Weidlich, T., Tessier, R., Paar, C.: HAL - the missing piece of the puzzle for hardware reverse engineering, trojan detection and insertion. IEEE Trans. Dependable Secur. Comput. **16**(3), 498–510 (2019). https://doi.org/10.1109/TDSC.2018.2812183
38. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.J.B.: The LED block cipher. In: Preneel, B., Takagi, T. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6917, pp. 326–341. Springer (2011). https://doi.org/10.1007/978-3-642-23951-9_22
39. Harttung, J.: Prince block cipher - vhdl implementation (2021), https://github.com/huljar/prince-vhdl/tree/master
40. Klix, S., Albartus, N., Speith, J., Staat, P., Verstege, A., Wilde, A., Lammers, D., Langheinrich, J., Kison, C., Sester, S., Holcomb, D.E., Paar, C.: Stealing maggie's secrets - on the challenges of IP theft through FPGA reverse engineering. CoRR **abs/2312.06195** (2023). https://doi.org/10.48550/arXiv.2312.06195
41. Lippmann, B., Werner, M., Unverricht, N., Singla, A., Egger, P., Dübotzky, A., Gieser, H.A., Rasche, M., Kellermann, O., Graeb, H.: Integrated flow for reverse engineering of nanoscale technologies. In: Shibuya, T. (ed.) Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019. pp. 82–89. ACM (2019). https://doi.org/10.1145/3287624.3288738
42. lowRISC Contributors: lowrisc: Collaborative open silicon engineering (2024), https://lowrisc.org/
43. lowRISC contributors: Open source silicon root of trust - opentitan (2024), https://opentitan.org/
44. lowRISC contributors: Opentitan earl grey chip datasheet (2024), https://opentitan.org/book/hw/top_earlgrey/doc/specification.html
45. Meade, T., Jin, Y., Tehranipoor, M.M., Zhang, S.: Gate-level netlist reverse engineering for hardware security: Control logic register identification. In: IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016. pp. 1334–1337. IEEE (2016). https://doi.org/10.1109/ISCAS.2016.7527495
46. Meijer, C., Bokslag, W., Wetzels, J.: All cops are broadcasting: TETRA under scrutiny. In: Calandrino, J.A., Troncoso, C. (eds.) 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023. pp. 7463–7479. USENIX Association (2023), https://www.usenix.org/conference/usenixsecurity23/presentation/meijer
47. Meijer, C., Moonsamy, V., Wetzels, J.: Where's crypto?: Automated identification and classification of proprietary cryptographic primitives in binary code. In: Bailey,

M.D., Greenstadt, R. (eds.) 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. pp. 555–572. USENIX Association (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/meijer
48. Miller, G.: The intelligence coup of the century. The Washington Post (2020), https://www.washingtonpost.com/graphics/2020/world/national-security/cia-crypto-encryption-machines-espionage/
49. Moradi, A., Barenghi, A., Kasper, T., Paar, C.: On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from xilinx virtex-ii fpgas. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011. pp. 111–124. ACM (2011). https://doi.org/10.1145/2046707.2046722
50. Moradi, A., Kasper, M., Paar, C.: Black-box side-channel attacks highlight the importance of countermeasures - an analysis of the xilinx virtex-4 and virtex-5 bitstream encryption mechanism. In: Dunkelman, O. (ed.) Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7178, pp. 1–18. Springer (2012). https://doi.org/10.1007/978-3-642-27954-6_1
51. Moradi, A., Schneider, T.: Improved side-channel analysis attacks on xilinx bitstream encryption of 5, 6, and 7 series. In: Standaert, F., Oswald, E. (eds.) Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9689, pp. 71–87. Springer (2016). https://doi.org/10.1007/978-3-319-43283-0_5
52. Mykhailo: Twitter post about microchips from a downed russian su-24m (2022), https://twitter.com/mxpoliakov/status/1606650167129788417
53. Narayanan, R.V., Venkatesan, A.N., Pula, K., Muthukumaran, S., Vemuri, R.: Reverse engineering word-level models from look-up table netlists. In: 24th International Symposium on Quality Electronic Design, ISQED 2023, San Francisco, CA, USA, April 5-7, 2023. pp. 1–8. IEEE (2023). https://doi.org/10.1109/ISQED57927.2023.10129373
54. National Institute of Standards and Technology (NIST): Data encryption standard (des). Tech. Rep. FIPS PUB 46-3, National Institute of Standards and Technology (NIST) (October 1999), https://csrc.nist.gov/files/pubs/fips/46-3/final/docs/fips46-3.pdf
55. National Institute of Standards and Technology (NIST): Secure hash standard (shs). Tech. Rep. FIPS PUB 180-4, National Institute of Standards and Technology (NIST) (August 2015). https://doi.org/10.6028/NIST.FIPS.180-4
56. Nohl, K., Evans, D., Starbug, Plötz, H.: Reverse-engineering a cryptographic RFID tag. In: van Oorschot, P.C. (ed.) Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA. pp. 185–194. USENIX Association (2008), http://www.usenix.org/events/sec08/tech/full_papers/nohl/nohl.pdf
57. Note, J., Rannaud, É.: From the bitstream to the netlist. In: Hutton, M., Chow, P. (eds.) Proceedings of the ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays, FPGA 2008, Monterey, California, USA, February 24-26, 2008. p. 264. ACM (2008). https://doi.org/10.1145/1344671.1344729
58. Oliscience: OpenCores, https://opencores.org
59. Pham, K.D., Horta, E.L., Koch, D.: BITMAN: A tool and API for FPGA bitstream manipulations. In: Atienza, D., Natale, G.D. (eds.) Design, Automation & Test in

Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017. pp. 894–897. IEEE (2017). https://doi.org/10.23919/DATE.2017.7927114
60. Primas, R.: Hardware design of ascon-128 and ascon-hash (v1.2) (2023), https://github.com/rprimas/ascon-verilog
61. Quadir, S.E., Chen, J., Forte, D., Asadizanjani, N., Shahbazmohamadi, S., Wang, L., Chandy, J.A., Tehranipoor, M.M.: A survey on chip to system reverse engineering. ACM J. Emerg. Technol. Comput. Syst. **13**(1), 6:1–6:34 (2016). https://doi.org/10.1145/2755563
62. Shi, Y., Ting, C.W., Gwee, B., Ren, Y.: A highly efficient method for extracting fsms from flattened gate-level netlist. In: International Symposium on Circuits and Systems (ISCAS 2010), May 30 - June 2, 2010, Paris, France. pp. 2610–2613. IEEE (2010). https://doi.org/10.1109/ISCAS.2010.5537093
63. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: An ultra-lightweight blockcipher. In: Preneel, B., Takagi, T. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6917, pp. 342–357. Springer (2011). https://doi.org/10.1007/978-3-642-23951-9_23
64. Strobel, D., Driessen, B., Kasper, T., Leander, G., Oswald, D.F., Schellenberg, F., Paar, C.: Fuming acid and cryptanalysis: Handy tools for overcoming a digital locking and access control system. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I. Lecture Notes in Computer Science, vol. 8042, pp. 147–164. Springer (2013). https://doi.org/10.1007/978-3-642-40041-4_9
65. Swierczynski, P., Fyrbiak, M., Koppe, P., Paar, C.: FPGA trojans through detecting and weakening of cryptographic primitives. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(8), 1236–1249 (2015). https://doi.org/10.1109/TCAD.2015.2399455
66. Swierczynski, P., Moradi, A., Oswald, D.F., Paar, C.: Physical security evaluation of the bitstream encryption mechanism of altera stratix II and stratix III fpgas. ACM Trans. Reconfigurable Technol. Syst. **7**(4), 34:1–34:23 (2015). https://doi.org/10.1145/2629462
67. Tajik, S., Lohrke, H., Seifert, J., Boit, C.: On the power of optical contactless probing: Attacking bitstream encryption of fpgas. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1661–1674. ACM (2017). https://doi.org/10.1145/3133956.3134039
68. TechInsights: TechInsights, https://www.techinsights.com
69. Texplained: Texplained, https://www.texplained.com
70. The Sage Developers: SageMath, the Sage Mathematics Software System (2024), https://www.sagemath.org
71. Torrance, R., James, D.: The state-of-the-art in IC reverse engineering. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings. Lecture Notes in Computer Science, vol. 5747, pp. 363–381. Springer (2009). https://doi.org/10.1007/978-3-642-04138-9_26
72. Werner, M., Lippmann, B., Baehr, J., Gräb, H.: Reverse engineering of cryptographic cores by structural interpretation through graph analysis. In: 3rd IEEE

International Verification and Security Workshop, IVSW 2018, Costa Brava, Spain, July 2-4, 2018. pp. 13–18. IEEE (2018). https://doi.org/10.1109/IVSW.2018.8494896

73. Ziener, D., Assmus, S., Teich, J.: Identifying FPGA ip-cores based on lookup table content analysis. In: Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL), Madrid, Spain, August 28-30, 2006. pp. 1–6. IEEE (2006). https://doi.org/10.1109/FPL.2006.311255