N. Faroß, S. Schwarz
Math Comput Sci(2025)19:7
https://doi.org/10.1007/s11786-025-00602-8

Mathematics in Computer Science

RESEARCH

Gröbner Bases for Boolean Function Minimization

Nicolas Faroß[1] and Simon Schwarz[2,3*]

*Correspondence: sschwarz@mpi-inf.mpg.de
[2]Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
Full list of author information is available at the end of the article

Communicated by Christopher W. Brown

Abstract

Boolean function minimization techniques try to find, for a given formula, a smaller equivalent formula. In this work, we present a novel technique for heuristic multi-level Boolean function minimization. By using an algebraic encoding, we embed the minimization problem into an algebraic domain, where algorithms for computing Gröbner bases become applicable. A Gröbner basis usually forms a compact representation of our encoded function. From the Gröbner basis, we then reconstruct an equivalent, more compact Boolean formula. The minimized formula is in the language of Boolean formulas with negation, conjunction, disjunction and exclusive-or operations. To the best of our knowledge, our approach is the first to use Gröbner bases for function minimization. We evaluate our approach on Boolean formulas created from arithmetic operations as well as on random formulas. Compared to state-of-the-art exact minimization algorithms, our approach can handle formulas with up to three times as many variables. Empirically, we obtain very compact formulas. In particular, our algorithm is especially efficient if there exists a comparatively small equivalent formula or if the minimized formula contains exclusive-or operations. In practice, such functions occur e.g. in embedded systems or cryptography. Overall, our approach forms a new, interesting trade-off between result minimality and runtime.

Keywords: Multi-level Logic Optimization, Boolean Function Synthesis, Gröbner Bases

Mathematics Subject Classification: 03B70, 13P10

1 Introduction

Boolean function minimization algorithms find, given a Boolean formula, a more compact equivalent Boolean formula. This is a central problem in e.g. circuit synthesis [1,2]. In general, the exact Boolean minimization problem is NP-hard; it is even Σ_2^P-complete [3,4]. Hence, exact approaches can quickly become infeasible. Thus, approaches in this area often have trade-offs between result quality and runtime. Previous work often focused on (heuristically) minimizing *disjunctive normal forms* (DNF) or used syntactical heuristics to simplify functions (Section 1.1).

Our contribution is a novel heuristic approach for multi-level Boolean function minimization. Given an input formula in DNF, our approach encodes it as an ideal over $\mathbb{F}_2$,

using a modification of previous encodings [5]. This representation allows the computation of a Gröbner basis with common algorithms [6]. The resulting Gröbner basis can then be interpreted again as an empirically smaller equivalent Boolean formula. This process is repeated recursively to further minimize the formula. Contrary to other minimization algorithms, our encoding over $\mathbb{F}_2$ can efficiently represent the exclusive-or ($\oplus$) operator, allowing exponentially more compact representations compared to DNF. Thus, our approach works particularly well for formulas which contain many exclusive-or operations. In practice, such formulas occur, for example, in microarchitectural hash functions [7], checksums [8] or basic cryptographic primitives [9]. In such settings, the definition of the function is often not given. Instead, it is possible to observe pairs of input and output. By constructing a DNF formula from the input-output pairs (Proposition 5) and minimizing it with our approach, we can *synthesize* such functions from observations. Our approach has been successfully used to reconstruct previously unknown microarchitectural hash functions [10]. Furthermore, our approach can also be used to synthesize functions from their specification, which has applications e.g. in reactive synthesis [11] or electronic hardware design [12].

We have implemented our minimization approach in SageMath [13], using SINGULAR [14], ESPRESSO [1], and GLPK [15] as libraries. In practice, we observe compact representations for many classes of formulas. We discuss our results on formulas that describe the result of arithmetic operations. In this case, we improve the resulting formula size by a factor of up to ten compared to the heuristic ESPRESSO minimizer. With a time limit of one hour, our approach is able to minimize arithmetic functions with up to 16 input bits, while state-of-the-art exact minimization techniques already time out with 8 input bits. Furthermore, we observe that we produce compact terms in a reasonable time even on random formulas. Moreover, for some classes of formulas, tight bounds are provable (Theorem 15). Empirically, the runtime of our algorithm correlates with the size of the minimized formula. Hence, in cases where a comparatively small formula is known to exist, our approach can minimize formulas with up to 20 input bits in minutes. Overall, our approach forms a new trade-off between runtime and result quality for multi-level logic minimization.

Parts of this work have been submitted as an extended abstract to the SC^2-workshop 2023 [16]. In addition to the workshop version, this version contains our set-cover optimization (Section 3.3), a full correctness proof (Theorem 13) as well as a generalization of Theorem 15. In addition, we evaluate our approach on arithmetic functions (Section 4.1) and include remarks on applications (Section 4.2). Last, this version contains all proofs and definitions as well as more detailed explanations.

This paper is organized as follows: First, we discuss related work on Boolean function minimization and Gröbner bases in Section 1.1. Next, we introduce our definitions in Section 2. In Section 3, we describe our approach and prove its correctness. Section 4.1 evaluates both the runtime and result size of our algorithm in comparison to other logic minimizers. Section 4.2 discusses applications of our algorithm. Last, Section 5 concludes this paper.

1.1 Related Work

Boolean Function Minimization Usually, a Boolean logic minimizer is given a formula in DNF, which can be easily obtained from a truth table. Then, it produces a small, equivalent formula. Existing tools can be classified by the shape of the output formula:

Two-level logic optimization tools produce again a formula in DNF. Classically, Quine and McCluskey [17,18] focus on optimal two-level minimization. However, their optimal approach can quickly become infeasible. Thus, later approaches such as ESPRESSO [1] make use of a heuristic search for producing a compact DNF. Still, representing a formula in DNF can have exponential overhead. For example, any DNF of $f(x_1, \ldots, x_n) = x_1 \oplus \cdots \oplus x_n$ is exponentially larger in n than f. This motivates more general minimization approaches like ours.

Multi-level logic optimization allows minimized formulas of arbitrary shape. Still, some tools produce only formulas of a fixed structure. For example, EXORCISM [19] always produces formulas of the shape "exclusive-sum-of-products". Other tools produce formulas of arbitrary depth and structure, for example MIS [20] or LSS [21]. Most multi-level optimization tools rely on two-level optimization and apply syntactic transformations such as *subexpression recognition* and replacement to the given formula [22]. More recently, approaches for exact multi-level synthesis based on satisfiability [23,24] have been studied. They provide optimal solutions but are only feasible for small instances. Last, generic syntax-guided synthesis approaches can be used to minimize logic expressions [25].

Our presented approach provides a heuristic method for multi-level logic optimization which uses results from computer algebra for logic minimization. In Section 4.1, we compare our approach with three existing logic minimizers: ESPRESSO, an exact algorithm based on satisfiability implemented in ABC, and a syntax-guided synthesis approach.

Gröbner Bases Gröbner Bases are widely used in computer algebra [6], for example for ideal membership testing of a polynomial. However, in this work, we focus on applications of Gröbner bases in logic. For example, Gröbner bases are used in the context of satisfiability checking and model counting, as well as in verification and SMT solving.

In satisfiability checking, Gröbner bases can be used for pre-processing clause sets in conjunctive normal form (CNF) [5,26]. Concretely, it is possible to encode a (sub-)set of clauses of a CNF formula as a system of polynomials. Then, a Gröbner basis for this system is computed. The resulting system of polynomials is then, again, interpreted as a set of clauses. The resulting, usually more compact set of clauses is *equivalent* to the original set. Hence, it is possible to replace sets of clauses with more compact, equivalent sets. Satisfiability checking on the pre-processed clauses is usually faster than on the original CNF [5]. However, the cost of computing the Gröbner bases usually outweighs the benefits gained in the satisfiability checking step. Moreover, this technique is computationally too expensive to perform on large instances, which are common in satisfiability checking. Thus, it is only applied on a heuristically chosen subset of clauses. A similar encoding can be used for model counting for Boolean formulas [27]. Given a set of clauses reduced by this technique, it becomes (computationally) easy to count satisfiable assignments. Our encoding of Boolean formulas to polynomials is similar to the encoding presented in the above two works. However, in contrast to the above approaches, our work does not deal with satisfiability checking, but is in the context of logic minimization. To this end, this work applies Gröbner bases to formulas in disjunctive normal form (DNF) instead of

applying it to formulas in CNF. Furthermore, our approach minimizes the whole formula, so no subsets are chosen for minimization.

Other applications of Gröbner basis are verification of arithmetic gates [28–30] and SMT solving over finite fields [31,32] or real numbers [33]. However, note that both applications do not directly encode Boolean formulas as polynomials, but use Gröbner bases for theory solving. Still, combined with our work, this suggests that Gröbner bases have multiple applications in logic and could be worthwhile to investigate further.

2 Preliminaries

Before we come to the main minimization algorithm in Section 3, we present some preliminaries from Boolean logic in Section 2.1 and computer algebra in Section 2.2.

2.1 Boolean Logic

From Boolean (or propositional) logic, we introduce the basic notions of formulas, equivalence and size of formulas as well as disjunctive normal forms.

Definition 1 (Syntax and Semantics of Boolean Formulas) Let $\Sigma = \{x_1, \dots, x_n\}$ be a set of Boolean variables. Then, the set of syntactically valid Boolean formulas (also called functions) $\mathrm{Prop}(\Sigma)$ is inductively defined by

$$\mathrm{false} \in \mathrm{Prop}(\Sigma), \qquad \mathrm{true} \in \mathrm{Prop}(\Sigma), \qquad x_i \in \mathrm{Prop}(\Sigma) \quad (\forall x_i \in \Sigma)$$

and

$$\begin{aligned} \neg\phi \in \mathrm{Prop}(\Sigma) \quad &(\text{negation}), & \phi \wedge \psi \in \mathrm{Prop}(\Sigma) \quad &(\text{conjunction}),\\ \phi \vee \psi \in \mathrm{Prop}(\Sigma) \quad &(\text{disjunction}), & \phi \oplus \psi \in \mathrm{Prop}(\Sigma) \quad &(\text{exclusive-or}) \end{aligned}$$

for $\phi, \psi \in \mathrm{Prop}(\Sigma)$.

The semantics of Boolean formulas are defined as follows: A Σ-valuation for variables is a map $\mathcal{A} : \Sigma \to \{0, 1\}$. A Σ-valuation $\mathcal{A}$ is inductively extended to assign each formula in $\mathrm{Prop}(\Sigma)$ a *truth value* from the set $\{0, 1\}$:

$$\begin{aligned} \mathcal{A}(\mathrm{false}) &:= 0, & \mathcal{A}(\mathrm{true}) &:= 1,\\ \mathcal{A}(\phi \wedge \psi) &:= \min(\mathcal{A}(\phi), \mathcal{A}(\psi)), & \mathcal{A}(\phi \vee \psi) &:= \max(\mathcal{A}(\phi), \mathcal{A}(\psi)),\\ \mathcal{A}(\neg\phi) &:= 1 - \mathcal{A}(\phi), & \mathcal{A}(\phi \oplus \psi) &:= \begin{cases} 1 & \text{if } \mathcal{A}(\phi) \neq \mathcal{A}(\psi),\\ 0 & \text{otherwise,} \end{cases} \end{aligned}$$

for $\phi, \psi \in \mathrm{Prop}(\Sigma)$. In the following, we also identify Σ-valuations $\mathcal{A}$ with elements $a \in \{0, 1\}^n$ via $\mathcal{A}(x_i) = a_i$ and we also write $\phi(a)$ instead of $\mathcal{A}(\phi)$.

Definition 2 (Equivalence for Boolean Formulas) We say that two Boolean formulas ϕ, ψ are (semantically) *equivalent* and write $\phi \equiv \psi$ if $\mathcal{A}(\phi) = \mathcal{A}(\psi)$ for all Σ-valuations $\mathcal{A}$.

Definition 3 (Literals, DNF) A *literal* is either a Boolean variable or its negation. Literals are usually denoted by L. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals, i.e. it is of shape $\bigvee_i \bigwedge_j L_{ij}$, where all L_{ij} are literals.

Definition 4 (Size of Boolean Formulas) For a Boolean formula $\phi \in \mathrm{Prop}(\Sigma)$, its size $\mathrm{size}(\phi)$ is inductively defined as follows:

$$\begin{aligned}
\mathrm{size}(x) &:= 1, \quad x \in \Sigma \\
\mathrm{size}(\mathrm{false}) &:= 1, & \mathrm{size}(\mathrm{true}) &:= 1, \\
\mathrm{size}(\phi \wedge \psi) &:= 1 + \mathrm{size}(\phi) + \mathrm{size}(\psi), & \mathrm{size}(\phi \vee \psi) &:= 1 + \mathrm{size}(\phi) + \mathrm{size}(\psi), \\
\mathrm{size}(\neg\phi) &:= 1 + \mathrm{size}(\phi), & \mathrm{size}(\phi \oplus \psi) &:= 1 + \mathrm{size}(\phi) + \mathrm{size}(\psi).
\end{aligned}$$

We also write $|\phi|$ to denote the size of ϕ.

For all Boolean formulas, Proposition 5 shows that there is an equivalent formula in DNF. In particular, this formula can easily be computed from a truth table representation of ϕ. Thus, a function specification in the form of a truth table is sufficient for our minimization technique.

Proposition 5 *Let ϕ be a Boolean formula in the variables $x_1, \ldots, x_n$. Then, we can construct an equivalent formula in DNF as follows:*

$$\phi \equiv \bigvee_{\substack{a \in \{0,1\}^n \\ \phi(a)=1}} \bigwedge_{i=1}^{n} L_{ai}, \qquad L_{ai} = \begin{cases} \neg x_i & \text{if } a_i = 0, \\ x_i & \text{if } a_i = 1. \end{cases}$$

Proof See for example the proof of [34, Theorem 15B]. □

Note that, in general, the size of the constructed DNF from Proposition 5 is exponential in the number of variables. Example 6 introduces a formula that will be used as a running example and demonstrates the translation from Proposition 5.

Example 6 Consider $\phi = (\neg y \vee (x \oplus z)) \wedge (x \vee z) \wedge (\neg x \vee \neg y \vee z) \in \mathrm{Prop}(\{x, y, z\})$. Its *truth table* looks as follows:

$\mathcal{A}(x)$	0	0	0	0	1	1	1	1
$\mathcal{A}(y)$	0	0	1	1	0	0	1	1
$\mathcal{A}(z)$	0	1	0	1	0	1	0	1
$\mathcal{A}(\phi)$	0	1	0	1	1	1	0	0

Following Proposition 5, we can construct an equivalent formula in disjunctive normal form:

$$\phi \equiv (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge \neg y \wedge z).$$

2.2 Polynomial Ideals and Gröbner Bases

Next, we introduce polynomial ideals and Gröbner bases. In this section, let k be any field and denote with $k[x_1, \ldots, x_n]$ the multivariate polynomial ring over k in the variables $x_1, \ldots, x_n$. In the following sections, we will then focus on the special case where $k = \mathbb{F}_2$ is given by the finite field with two elements.

Definition 7 (Ideals) An *ideal* I in the polynomial ring $k[x_1, \ldots, x_n]$ is a subset $I \subseteq k[x_1, \ldots, x_n]$ which is of the form

$$I = \left\{ \sum_{i=1}^{m} f_i g_i \mid f_1, \ldots, f_m \in k[x_1, \ldots, x_n] \right\}$$

for polynomials $g_1, \ldots, g_m \in k[x_1, \ldots, x_n]$. The polynomials g_i are called generators of I, the set $\{g_i\}$ is called a *generating set* of I.

Gröbner bases are special sets of generators of an ideal, which can be used to perform multivariate polynomial division. They are widely used in computer algebra [6] and allow for example ideal membership testing or the computation of the zeros of a system of polynomial equations. Although the exact definition will not be used in the rest of the paper, we give the definition of Gröbner bases for completeness. For further information on Gröbner bases, we refer again to [6].

In the following, we fix an admissible ordering on $k[x_1, \ldots, x_n]$, for example the lexicographic ordering induced by $x_1 > \cdots > x_n$. Using this ordering, we can define the leading term $\mathrm{LT}(f)$ of a polynomial $f \in k[x_1, \ldots, x_n]$ as its largest monomial with its corresponding coefficient. Assume g is another polynomial such that $\mathrm{LT}(g)$ divides the leading term $\mathrm{LT}(f)$. Then, we can perform the reduction step $f \to f - \frac{\mathrm{LT}(f)}{\mathrm{LT}(g)} \cdot g$ to eliminate the leading term of f. We say a polynomial f can be reduced to a polynomial f' by a set of polynomials G if there exists a sequence of reduction steps starting from f and resulting in f' where each reduction is performed by an element of G. Using this notion of reduction, we can now define Gröbner bases.

Definition 8 (Gröbner Basis) Let $I \subseteq k[x_1, \ldots, x_n]$ be a ideal and $G \subseteq I$ a finite subset with $0 \notin G$. Then G is a *Gröbner basis* if every element in I can be reduced to 0 using elements in G.

One can show that a Gröbner basis is a generating set for the ideal I and that a polynomial f lies in I if and only if it reduces to 0 with respect to G. Further, Hilbert's basis theorem implies that every ideal has a finite Gröbner basis. However, Gröbner bases are not unique and fail to be minimal in general, which leads to the following definition.

Definition 9 (Reduced Gröbner Basis) Let G be a Gröbner basis. Then G is called *reduced* if every element of G has the leading coefficient 1 and can not be further reduced using any other element in G.

A reduced Gröbner basis is a uniquely determined generating set of the ideal I and does only depend on the fixed monomial ordering. Furthermore, it is possible to compute a (reduced) Gröbner basis from a generating set $\{f_1, \ldots, f_m\}$ in double exponential time.

3 Gröbner Bases for Logic Minimization

Given a Boolean formula ϕ in n variables, we want to find an equivalent formula of smaller size. Our idea is to transform ϕ to a corresponding ideal I in the polynomial ring $\mathbb{F}_2[x_1, \ldots, x_n]$. Then, a Gröbner basis of I yields a formula that is equivalent to ϕ and empirically smaller. First, we outline the relationship between Boolean formulas and polynomials of $\mathbb{F}_2$, before we present our main algorithm.

3.1 Boolean Formulas and Polynomials

Consider the polynomial ring over the finite field $\mathbb{F}_2 = \{0, 1\}$. Note that multiplication $x \cdot y$ on $\mathbb{F}_2[x, y]$ aligns with $x \wedge y$ in $\mathrm{Prop}(\{x, y\})$. Furthermore, addition $x + y$ aligns with

the exclusive-or operation $x \oplus y$. Hence, we can identify a polynomial

$$f = \sum_i \prod_j y_{ij} \in \mathbb{F}_2[x_1, \ldots, x_n]$$

where $y_{ij} \in \{0, 1, x_1, \ldots, x_n\}$ with a Boolean formula of the form

$$\phi = \bigoplus_i \bigwedge_j y_{ij} \in \mathrm{Prop}(\{x_1, \ldots, x_n\}).$$

Following the usual convention, we syntactically identify the values 0 and 1 in $\mathbb{F}_2$ with the Boolean formulas false and true, respectively. Semantically, it holds that $f(a) = \phi(a)$ for all $a \in \{0, 1\}^n = \mathbb{F}_2^n$.

We can represent any Boolean formula ϕ by an equivalent polynomial over $\mathbb{F}_2$ by expressing additional connectives in terms of $\wedge$ and $\oplus$. For example, we have $\neg x = x \oplus 1$ and $x \vee y = \neg(\neg x \wedge \neg y)$, which will be represented in $\mathbb{F}_2$ as $x + 1$ and $1 + (1 + x) \cdot (1 + y) = xy + x + y$, respectively.

The previous relationship between Boolean formulas and polynomials can be further extended to a correspondence between equivalent formulas and (radical) ideals in $\mathbb{F}_2[x_1, \ldots, x_n]$ by showing the zero-set of an ideal corresponds exactly to non-satisfying assignments of the Boolean formula. Since this result relies on facts from algebraic geometry over finite fields, we also present a proof for completeness.

Proposition 10 *There is a bijection between equivalence classes of Boolean formulas* $[\phi]$ *in n variables and ideals* $I \subseteq \mathbb{F}_2[x_1, \ldots, x_n]$ *containing* $x_1^2 + x_1, \ldots, x_n^2 + x_n$, *such that*

$$\phi(x) = 0 \quad \Longleftrightarrow \quad \forall f \in I, f(x) = 0.$$

Proof Let $[\phi]$ be an equivalence class of Boolean formulas in n variables. Then it is uniquely determined by its zero set $\{x \in \mathbb{F}_2^n \mid \phi(x) = 0\}$. Conversely, any set $Y \subseteq \mathbb{F}_2^n$ is the zero set of the formula $\bigwedge_{y \in Y} \bigvee_{i=1}^n (x_i \oplus y_i)$. Hence, there is a bijection between equivalent formulas and subsets of $\mathbb{F}_2^n$. Similarly, any ideal containing $x_1^2 + x_1, \ldots, x_n^2 + x_n$ is uniquely determined by its zero set $\{x \in \mathbb{F}_2^n \mid \forall f \in I, f(x) = 0\}$ by Hilbert's Nullstellensatz for finite fields [35]. Additionally, every subset $Y \subseteq \mathbb{F}_2^n$ is finite and hence a zero set of an ideal by elementary results from algebraic geometry. Further, this ideal can always be assumed to contain $x_1^2 + x_1, \ldots, x_n^2 + x_n$ since these polynomials vanish on $\mathbb{F}_2^n$. Thus, we have a bijection between ideals containing $x_1^2 + x_1, \ldots, x_n^2 + x_n$ and subsets $Y \subseteq \mathbb{F}_2^n$. By combining the previous two bijections, we obtain

$$\phi \longleftrightarrow \{x \mid \phi(x) = 0\} \longleftrightarrow \{x \mid \forall f \in I, f(x) = 0\} \longleftrightarrow I,$$

which gives a one-to-one correspondence between equivalent Boolean formulas ϕ and ideals I containing $x_1^2 + x_1, \ldots, x_n^2 + x_n$. □

Note that the previous proposition can alternatively be formulated for the ring of boolean polynomials $R = \mathbb{F}_2[x_1, \ldots, x_n]/(x_1^2 + x_1, \ldots, x_n^2 + x_n)$, which would yield a bijection between equivalent boolean formulas and arbitrary ideals in R. However, we explicitly work over the ring $\mathbb{F}_2[x_1, \ldots, x_n]$ in order to avoid the computation of Gröbner bases over quotient rings.

Importantly, we can explicitly convert any Boolean formula to generators of the corresponding ideal and vice versa. If the Boolean formula ϕ is already in the form of a polynomial, then the previous proposition implies that the corresponding ideal is generated by ϕ and $x_1^2 + x_1$, ..., $x_n^2 + x_n$. In case of an arbitrary Boolean formula, it is always possible to first convert it into DNF as in Proposition 5. Then we can use Proposition 10 to show that a generating set is given as follows.

Lemma 11 *Let $\phi = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{k_i} L_{ij}$ be a formula in n variables and DNF. Then the corresponding ideal in Proposition 10 is generated by $x_1^2 + x_1, \ldots, x_n^2 + x_n$ and $g_i = \prod_{j=1}^{k_i} L_{ij}$ for all $1 \leq i \leq m$, where we identify the literal x_ℓ with the polynomial x_ℓ and the literal $\neg x_\ell$ with the polynomial $x_\ell + 1$.*

Conversely, given a generating set of an ideal, we can construct a Boolean formula from it. Again, the following lemma follows directly from Proposition 10.

Lemma 12 *Let $I \subseteq \mathbb{F}_2[x_1, \ldots, x_n]$ be an ideal which is generated by $g_1, \ldots, g_m$ and $x_1^2 + x_1, \ldots, x_n^2 + x_n$. Then a corresponding Boolean formula in the sense of Proposition 10 is given by $\phi = \bigvee_{i=1}^{m} g_i$, where we identify the polynomials g_i with Boolean formulas.*

Finally, let us comment on the role of the polynomials $x_i^2 + x_i$. In Boolean formulas, they represent the idempotency law $x_i \wedge x_i = x_i$. From an abstract point of view, adding the generators $x_i^2 + x_i$ to an ideal over $\mathbb{F}_2$ yields its radical, see [35]. On the other hand, the polynomials $x_i^2 + x_i$ allow the elimination of all higher powers of x_i during a Gröbner basis computation. Thus, there exist only 2^n different leading monomials besides x_i^2, such that the size of a reduced Gröbner basis is bounded by $2^n + n$. Therefore, it is possible to compute a Gröbner basis in time $2^{O(n)}$ using Buchberger's algorithm, see Proposition 4.1.1 in [27]. This provides a significant improvement compared to the usual double exponential bound for Gröbner bases.

3.2 Gröbner Basis Minimization

Empirically, Gröbner bases are relatively small if the ideal is not too complex. Hence, we can simplify a Boolean formula by converting it to an ideal I. Then, we compute a reduced Gröbner basis of I to obtain a set of generators G, which are then converted back to an equivalent formula of the form $\bigvee_{g \in G} g$. Since $\bigvee_{g \in G} g$ is again equivalent to $\bigvee_{g \in G} \neg\neg g$, we can apply this approach recursively to the terms $\neg g$ to further reduce the size and obtain a formula of the form $\bigvee_i \neg \bigvee_j \neg \bigvee_k \ldots \cong \bigvee_i \bigwedge_j \bigvee_k \ldots$. This allows the nesting of conjunctions and disjunctions, which enables our approach to perform multi-level logic minimization. Before we present the pseudo-code of our main algorithm, we first introduce the following subroutines.

1. GröbnerBasis($f_1, \ldots, f_m$): Return a reduced Gröbner basis of the ideal generated by the polynomials $f_1, \ldots, f_m$. In our implementation all Gröbner bases are computed with respect to the degree reverse lexicographic order.
2. Ideal(ϕ): Return generators of the ideal corresponding to ϕ as described in Lemma 11. The formula ϕ has to be given in DNF.
3. Formula(f): Return the Boolean formula corresponding to the polynomial f by replacing the operation $+$ and $\cdot$ with $\oplus$ and $\wedge$ respectively.
4. DNF(ϕ): Return a formula in DNF which is equivalent to ϕ, see Proposition 5.

Using these subroutines, the Gröbner basis minimization algorithm is described in Algorithm 1.

Algorithm 1 Recursive Gröbner Minimization

```
1: function MINIMIZE(φ)
2:     G ← GRÖBNERBASIS(IDEAL(φ)) \ {x_1^2 + x_1, ..., x_n^2 + x_n}
3:     for g ∈ G do
4:         if g is linear or recursion limit is reached then
5:             φ_g ← FORMULA(g)
6:         else
7:             φ_g ← ¬MINIMIZE(DNF(¬FORMULA(g)))
8:         end if
9:     end for
10:    return ⋁_{g∈G} φ_g
11: end function
```

Note that in our implementation (Section 4), the recursion limit is set to 6. Before we discuss the size of the returned formula in the following, we quickly prove the correctness of the algorithm using the considerations in Section 3.1.

Theorem 13 *Let ϕ be a Boolean formula in DNF. Then Algorithm 1 outputs a Boolean formula which is equivalent to ϕ.*

Proof By the definition of IDEAL(ϕ) and Lemma 11, the Gröbner basis of IDEAL(ϕ) generates the ideal corresponding to the formula ϕ in the sense of Proposition 10. Thus, Lemma 12 implies that $\bigvee_{g\in G} \phi_g$ is equivalent to the original formula ϕ, where we identify the polynomial g with the formula FORMULA(g). If the termination criterion is not satisfied, we can replace g with $\neg\neg g$, which is again equivalent to $\neg$DNF($\neg g$). By applying MINIMIZE to DNF($\neg g$), it follows inductively that g is equivalent to $\neg$ MINIMIZE(DNF($\neg g$)). Therefore, the algorithm returns a formula that is equivalent to the original ϕ. Since not every formula results in linear polynomials, termination is guaranteed by using a fixed recursion limit. □

Example 14 Recall the formula ψ established in Example 6, which is given by

$$\psi = (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge \neg y \wedge z).$$

According to Lemma 11, the ideal I corresponding to ψ in the sense of Proposition 10 is generated by

$$(x+1)\cdot(y+1)\cdot z, \quad (x+1)\cdot y\cdot z, \quad x\cdot(y+1)\cdot(z+1), \quad x\cdot(y+1)\cdot z,$$
$$x^2+x, \qquad y^2+y, \qquad z^2+z.$$

The reduced Gröbner basis of this ideal is given by

$$\text{GRÖBNERBASIS}(I) = \{xy+x,\ xz+z,\ yz+z,\ x^2+x,\ y^2+y,\ z^2+z\}.$$

By removing the polynomials x^2+x, y^2+y, z^2+z and following Lemma 12, we can construct the formula

$$\chi' = ((x \wedge y) \oplus x) \vee ((x \wedge z) \oplus z) \vee ((y \wedge z) \oplus z)$$

from this Gröbner basis. By applying the minimization algorithm recursively, we obtain the simpler formula

$$\chi = (x \wedge \neg y) \vee (\neg x \wedge z) \vee (\neg y \wedge z)$$

as the output of Algorithm 1. By Theorem 13, it holds that $\chi \equiv \psi$.

Note that it is not guaranteed that our algorithm produces a smaller formula. By the previous bound, the Gröbner bases can have a size exponential in the number of variables. However, we empirically observed very compact formulas, see Section 4. In particular, if a small Boolean formula exists, the result of Algorithm 1 is empirically small as well, see Section 4.1. Furthermore, if the found Gröbner basis is small, its computation is significantly faster. Hence, for such formulas, our approach can handle up to 20 input bits within a reasonable time.

Furthermore, for special classes of formulas, tight bounds on the result size can be proven. For example, we obtain the following upper bound on the size of disjuncts of linear formulas.

Theorem 15 *Let ϕ be a formula in DNF which is equivalent to $\bigvee_{i=1}^{m} \bigoplus_{j=1}^{n} A_{i,j}x_j \oplus b_i$ for a matrix $A \in \mathbb{F}_2^{m\times n}$ and a vector $b \in \mathbb{F}_2^m$. Then $|MINIMIZE(\phi)| < 2 \cdot n \cdot \operatorname{rank}(A|b)$, where $A|b \in \mathbb{F}_2^{m\times(n+1)}$ is obtained by adding b as a column to A.*

Proof Consider a Boolean formula ϕ which is equivalent to $\bigvee_{i=1}^{m} \bigoplus_{j=1}^{n} A_{i,j}x_j \oplus b_i$ for a matrix $A \in \mathbb{F}_2^{m\times n}$ and a vector $b \in \mathbb{F}_2^m$. Then the corresponding ideal I from Proposition 10 is generated by $x_i^2 + x_i$ and the linear terms $\sum_{j=1}^{n} A_{i,j}x_i + b_i$ defined by the rows of $A|b$. Denote with r the rank of $A|b$ and let $C \in \mathbb{F}_2^{r\times(n+1)}$ be the matrix given by the non-zero rows in the reduced row echelon form of $A|b$, which can be computed using Gaussian elimination. Then one can check that a reduced Gröbner basis of I is given by the terms $\sum_{j=1}^{n} C_{i,j}x_i + C_{i,j+1}$ corresponding to the rows of C and the terms $x_i^2 + x_i$ for which x_i is not a leading monomial of one of the first terms. Since a reduced Gröbner basis is unique, it will also be computed by our minimization algorithm. Further, the terms of the form $x_i^2 + x_i$ are redundant and will be removed, such that a formula of the form $\bigvee_{i=1}^{r} \bigoplus_{j=1}^{n} C_{i,j}x_j \oplus C_{i,j+1}$ is returned. By counting the number of operators and variables, we obtain that the size of such a formula is always less than $2nr$. □

3.3 Set Cover Post-Processing

In the following, we present an additional post-processing step, which can further reduce the size of the formula produced by the Gröbner basis minimization algorithm. Consider the output of Algorithm 1, which is of the form $\bigvee_{g\in G} \phi_g$ for a set of polynomials G. Recall that ϕ_g is the Boolean formula corresponding to the polynomial g. In general, there can exist a proper subset $H \subseteq G$ which yields an equivalent formula, i.e. the subset satisfies $\bigvee_{h\in H} \phi_h(x) = \bigvee_{g\in G} \phi_g(x)$ for all $x \in \mathbb{F}_2^n$. Note that H does not necessarily generate the same ideal as G but only covers the same set of non-zeros.

Finding such a subset $H \subseteq G$ of small polynomials that yield an equivalent formula can be formulated as a weighted set-cover problem. Define the sets $S_g = \{x \in \mathbb{F}_2^n \mid \phi_g(x) = 1\}$ for all $g \in G$ and the universe $U = \bigcup_{g\in G} S_g$. Then, we want to find the subset $H \subseteq G$

that solves the optimization problem

$$\text{minimize} \sum_{h \in H} |\phi_h| \quad \text{subject to} \bigcup_{h \in H} S_h = U,$$

where $|\phi_h|$ denotes the size of the formula ϕ_h. The correctness of this approach follows from the fact that

$$\bigvee_{h \in H} \phi_h \equiv \bigvee_{g \in G} \phi_g \quad \Longleftrightarrow \quad \bigcup_{h \in H} S_h = U.$$

This weighted set-cover problem can directly be translated into an integer linear program using the approach in [36]. For each $g \in G$, we introduce a binary integer variable $z_g \in \{0, 1\}$ which indicates if the element g belongs to the subset H. By solving the linear integer optimization problem

$$\text{minimize} \sum_{g \in G} |\phi_g| \cdot z_g \quad \text{subject to} \sum_{\substack{g \in G \\ g(x)=1}} z_g \geq 1 \quad \forall x \in U,$$

we obtain a set $H = \{g \in G \mid z_g = 1\}$ which yields a small equivalent formula of the form $\bigvee_{h \in H} \phi_h$. In general, note that integer linear programming is NP-complete and current exact algorithms have a worst case complexity exponential in the number of variables. However, we do not have to solve the minimization problem exactly and approximate solutions will already result in smaller formulas compared to the original one. In practice, this step does not dominate the runtime of the overall approach. Further, this post-processing can automatically remove the polynomials $x_i^2 + x_i$ which were removed manually in Algorithm 1.

Example 16 Recall the formula $\phi = (x \wedge \neg y) \vee (\neg x \wedge z) \vee (\neg y \wedge z)$ computed by Algorithm 1 in Example 14. We can split ϕ into the following parts:

$$\phi_f = x \wedge \neg y, \quad \phi_g = \neg x \wedge z, \quad \phi_h = \neg y \wedge z$$

such that $\phi = \phi_f \vee \phi_g \vee \phi_h$. Now, consider the following table, listing the evaluation of the sub-formulas ϕ_f, ϕ_g and ϕ_h.

$\mathcal{A}(x)$	0	0	0	0	1	1	1	1
$\mathcal{A}(y)$	0	0	1	1	0	0	1	1
$\mathcal{A}(z)$	0	1	0	1	0	1	0	1
$\mathcal{A}(\phi_f)$	0	0	0	0	**1**	**1**	0	0
$\mathcal{A}(\phi_g)$	0	**1**	0	**1**	0	0	0	0
$\mathcal{A}(\phi_h)$	0	1	0	0	0	1	0	0
$\mathcal{A}(\phi)$	0	1	0	1	1	1	0	0

Note that ϕ_f and ϕ_g already cover all valuations where ϕ is true. Hence, we have $\phi \equiv \phi_f \vee \phi_g \equiv (x \wedge \neg y) \vee (\neg x \wedge z)$. The latter formula is found by our set-cover optimization.

3.4 DNF Pre-Processing

In addition to our main algorithm and the post-processing step, we use the following improvement: After converting a formula into DNF, Proposition 5, we can use existing two-level minimization algorithms on the resulting DNF. Such algorithms reduce the size

of the DNF, which yields a smaller generating set of the corresponding ideal. Thus, this step can significantly speed up the subsequent Gröbner basis computation. Note that the corresponding ideal stays the same. Hence, the resulting Gröbner basis is not changed, as it is unique for a given monomial order and does not depend on a particular generating set. Thus, the final minimized formula also remains stable under this optimization. In practice, this pre-processing step has a negligible runtime.

4 Evaluation

Our algorithm has been implemented in SageMath [13], using SINGULAR [14] for computing Gröbner bases, as well as ESPRESSO [1] for DNF pre-processing and GLPK [15] for set-cover computations. A prototype implementation is publicly available at [37].

4.1 Comparison with Existing Tools

In the following, we compare our results to the results achieved by other Boolean function minimizers. In particular, we compare against the following four tools:

- ESPRESSO [1] is the most well-known tool for two-level minimization. Note that the resulting formula of ESPRESSO is always a DNF.
- ABC's 'exact' command [23,38], which uses an encoding to SAT to exactly minimize Boolean formulas. Formula minimization is limited to 7 input bits. This approach should produce optimal results. [1]
- cvc5 [39] implements a syntax-guided synthesis (SyGuS) [25] approach. To this end, we establish a specification in SyGuS format [40]. This approach produces optimal results.
- A counter-example guided inductive synthesis (CEGIS) approach [41]. This approach again produces optimal results.

Recall that our approach produces formulas that adhere to the syntax defined in Definition 1. In particular, the formulas can contain negations, disjunctions, conjunctions and exclusive-or operators. SyGuS and the CEGIS approach produce formulas over the same alphabet. In contrast, ESPRESSO can only produces disjunctive normal forms. All tools were run on Linux on a single core of an AMD EPYC 7702 2, 2GHz processor. Each run had a limit of 32GB of memory.

Arithmetic Functions First, we compare results on Boolean formulas which are constructed from arithmetic functions. Such functions often occur in practice, for example during the optimization of circuits for an arithmetic-logic unit in a microprocessor. Let $n \geq 1$ and consider the $2n$ Boolean variables $x_1, \ldots, x_{2n}$. Furthermore, let $[a_k \ldots a_1]$ denote the natural number with binary digits $a_k, \ldots, a_1$ and most significant bit a_k. Let $(\cdot)_n$ be a function that extracts the n-th least significant bit, where $(\cdot)_1$ denotes the least significant bit. Hence, $([a_k \ldots a_1])_n = a_n$.

For evaluation of our approach, we consider an exemplary Boolean function for addition and multiplication, respectively. In both functions, we take the n-th bit of the result.

[1] Note that the current version of ABC does not produce minimal solutions on some inputs. We suspect that this is due to a small bug in their implementation.

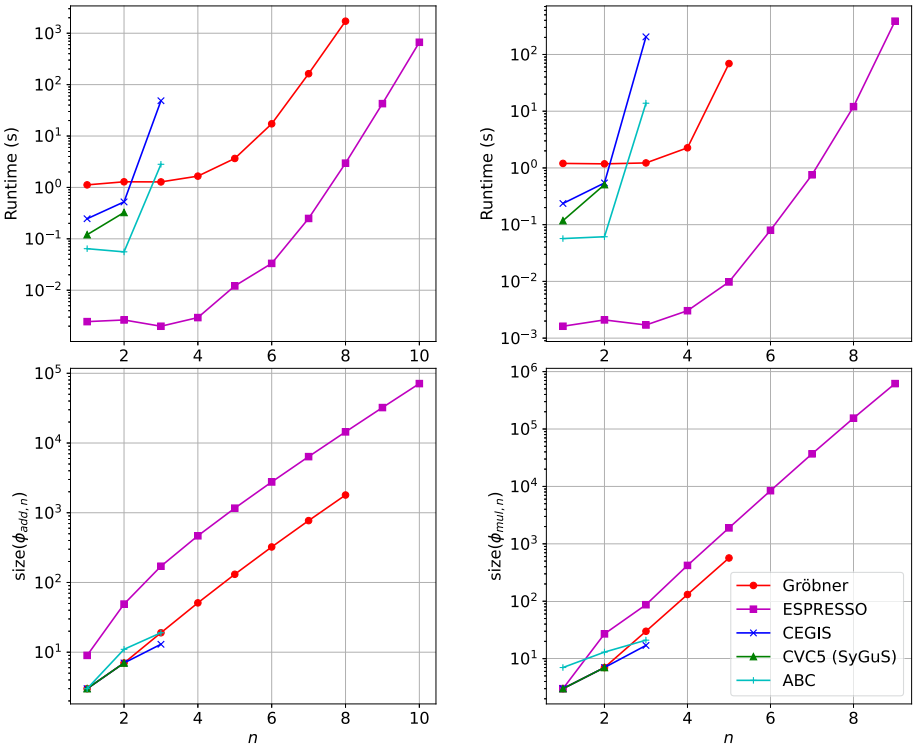

Fig. 1 Runtime and resulting formula size for $\phi_{add,n}$ (left) and $\phi_{mul,n}$ (right). The size of the resulting formula is displayed at the bottom, the runtime at the top. Note that all vertical axes are logarithmic

Table 1 Comparison of the constructed formulas for the first $\phi_{add,n}$. Note that the minimal formula for $\phi_{add,3}$ is not found by CVC5 within the 1h time limit.

Formula	Gröbner [Size]		Minimal Formula [Size]	
$\phi_{add,1}$	$x_1 \oplus x_2$	[3]	$x_1 \oplus x_2$	[3]
$\phi_{add,2}$	$(x_1 \wedge x_3) \oplus x_2 \oplus x_4$	[7]	$(x_1 \wedge x_3) \oplus x_2 \oplus x_4$	[7]
$\phi_{add,3}$	$(x_1 \wedge x_2 \wedge x_4) \oplus (x_1 \wedge x_4 \wedge x_5)$ $\oplus (x_2 \wedge x_5) \oplus x_3 \oplus x_6$	[19]	$(x_1 \wedge x_4 \wedge (x_2 \oplus x_5))$ $\oplus (x_3 \oplus x_6 \oplus (x_2 \wedge x_5))$	[15]

Formally, the Boolean functions that we consider for our benchmarks are:

$$\phi_{add,n} \equiv ([x_n \dots x_1] + [x_{2n} \dots x_{n+1}])_n,$$
$$\phi_{mul,n} \equiv ([x_n \dots x_1] \cdot [x_{2n} \dots x_{n+1}])_n.$$

For all of those formulas, we create a formula in DNF (Proposition 5) and pass it to the respective logic minimizer. In general, the created DNF is of exponential size in n. Fig. 1 shows the sizes of the resulting, minimized formulas and the runtime of all approaches. All programs had a time limit of one hour for each formula. Omitted points mean that the program did not terminate within the time limit. For $\phi_{add,n}$, our approach can handle up to $n = 8$ (16 input bits) within 1 hour, while CVC5 with the exact SyGuS approach already times out for $n = 3$, and ABC cannot handle cases with $n \geq 4$. Compared to ESPRESSO, we achieve an almost constant factor of 10 times smaller formulas. Still, our result size seems to scale exponentially, even though polynomial-size formulas exist [42]. Table 1 shows the first three formulas reconstructed by our approach in comparison with the optimal formulas.

Table 2 Number of solved benchmarks within the time limit of 1 minute for all K and n. For each K and n, there are 100 randomly sampled formulas

K	$n \rightarrow$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	Gröbner	100	100	100	100	100	100	100	100	100	100	100	100	100	100	91	72	38	16	12	4
	ESPRESSO	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	95	80	62	42	34
	CVC5 (SyGuS)	100	100	100	100	87	50	33	23	0	0	0	0	0	0	0	0	0	0	0	0
	CEGIS	100	100	100	100	98	81	73	46	30	29	11	5	2	2	0	0	0	0	0	0
	ABC	-	81	94	96	97	93	95	-	-	-	-	-	-	-	-	-	-	-	-	-
10	Gröbner	100	100	100	100	100	100	100	100	100	99	98	98	91	83	68	39	22	8	8	3
	ESPRESSO	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	82	63	45	27
	CVC5 (SyGuS)	100	100	100	92	69	35	22	21	6	9	8	5	5	0	2	1	2	0	0	0
	CEGIS	100	100	100	100	92	61	37	33	15	15	11	4	2	0	2	1	2	0	0	0
	ABC	-	72	85	91	94	75	65	-	-	-	-	-	-	-	-	-	-	-	-	-

For $\phi_{mul,n}$, we can compute formulas up to $n = 5$ within the time limit of one hour. Here, our minimized formulas are smaller by an almost constant factor of approximately 4 in comparison to ESPRESSO.

Random Compact Formulas Second, we evaluate our algorithm on a set of random but compact formulas. The benchmark formulas were generated as follows: Let n be the number of variables in our formula, and let $\Sigma = \{x_1, \ldots, x_n\}$. We then create a set of formulas $B_K = \{\phi \mid \phi \in \text{Prop}(\Sigma) \wedge \text{size}(\phi) = K \cdot n\}$. The formulas in B_K are linear in size in the amount of variables.

We evaluate our algorithm for $K = 3$ and $K = 10$. The first class are compact formulas where each variable, on average, occurs about two times in the formula. In the second class, the formulas are bigger and, on average, each variable occurs about 5 times. For evaluation, we sample 100 formulas uniformly at random from B_K. For all of those formulas, we create a DNF (Proposition 5) and pass it to the respective tool. For each run of one tool on one formula, we set a time limit of 1 minute.

Table 2 shows the number of benchmarks that are solved within the time limit by each solver for $K = 3$ and $K = 10$. For $K = 3$, note that our performance is only slightly worse than ESPRESSO. However, for $K = 10$ we time out on significantly more formulas. Hence, observe that our approach performs better if a small, minimized formula exists.

Figure 2 compares the average output size of the formulas for $K = 3$ and $K = 10$. To not skew results, a point is only plotted if the respective solver solved all 100 random formulas for a given K and n. If $K = 3$, we generate significantly better results, with about half the size of ESPRESSO. For $K = 10$, we only generate formulas about 20 percent smaller.

Overall, we conclude that our algorithm performs well on random formulas that have a small size compared to their number of variables. This holds for both the runtime and the result formula size of our approach. If the size of the formula grows, the improvement of our results in comparison to ESPRESSO diminishes.

4.2 Applications

In practice, the goal is not to minimize random functions but functions for a specific application with a very specific structure. The previous section shows promising results in particular if a small Boolean formula exists. Furthermore, due to our encoding in $\mathbb{F}_2$, our approach performs well if the resulting function contains many exclusive-or operations. Hence, applications that provide formulas with such structure are well-suited for our algorithm.

Such functions arise, for example, in microarchitectural hash functions, as outlined in the first case study. Furthermore, such functions are used in checksums and error-correcting codes such as the "Cyclic Redundancy Check" (CRC), see the second case

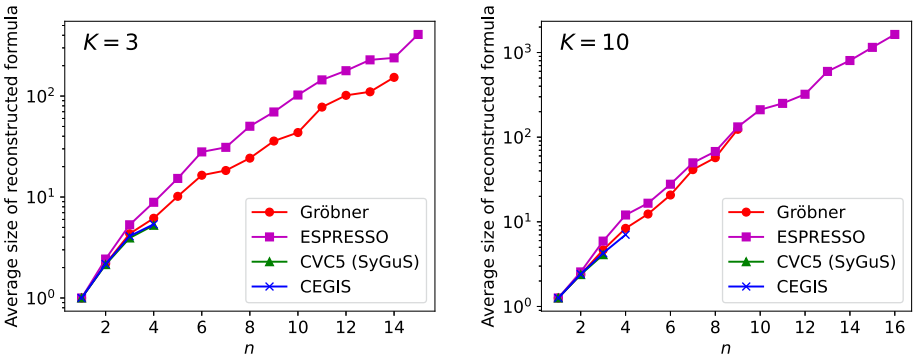

Fig. 2 Average size of the reconstructed formulas for $K = 3$ (left) and $K = 10$ (right). Note that points are only included if the respective solver solved all 100 randomly sampled functions. The vertical axis of both plots is log-scaled

study. Moreover, functions that use exclusive-or operations are common in cryptography [9].

Case Study: Microarchitectural Hash Functions Our approach was successfully used for reverse engineering microarchitectural hash functions. Current processors cache each memory address in a specific cache line. As there are only limited cache lines, hash values are computed from the memory addresses, which then determine the corresponding cache lines. These hash values are often computed by hash functions that extensively use exclusive-or operations.

Such hash functions are usually hard-wired inside a processor and not made public. Usually, they are transient in normal execution. However, they play an important rule in side-channel analysis. More details, including details on how truth tables for microarchitectural hash functions are sampled, can be found in the dedicated publication [10].

Case Study: Checksums in Embedded Systems When transmitting data in embedded systems, often a checksum or error-correcting code is appended to a transmission. In proprietary systems, the checksum algorithm is usually not public knowledge. Still, understanding the checksum algorithms can be useful for e.g. testing of the system, as it allows computing checksums of modified data.

Common checksum algorithms include simple parity checks or more generally cyclic redundancy checks (CRCs) [43]. Mathematically, these checksums are obtained as the remainder of a polynomial division in the ring $\mathbb{F}_2[x]$, see [8]. Thus, the checksum depends affine linearly on a fixed set of input bits. According to Theorem 15, our algorithm can find a compact Boolean formula for each checksum bit. This operation is feasible in practice, since many protocols like USB, GSM and Bluetooth use CRC checksums with a block size of up to 16 bits [44].

Note that affine linear functions can be reconstructed with simpler methods. However, the power of our approach lies in its flexibility. We do not need prior guarantees that an unknown function is affine linear. This makes our algorithm suitable for generic black-box function recovery while still maintaining guarantees for the linear case. In particular, note that most functions in embedded systems are not more complex as necessary, as simple functions are cheaper in realization. Hence, our approach is especially suitable for black-box recovery of Boolean functions from embedded systems.

5 Conclusion

We have presented a novel technique for heuristic multi-level Boolean formula minimization based on an algebraic encoding of formulas, Gröbner basis computations and a set-cover reduction. Our approach forms a new, interesting trade-off between result minimality and runtime. Currently, our algorithm can handle functions with up to 18 input bits, whereas exact synthesis can only handle up to about 7 input bits within minutes. Empirically, our approach produces significantly smaller formulas than two-level minimization algorithms such as ESPRESSO. In particular, our approach works well in two cases: First, our algorithm produces compact results comparatively fast if a small equivalent formula exists. Second, our approach performs well if the resulting minimized formula has many exclusive-or operations.

Our paper shows a novel application of Gröbner basis computations in logic minimization. In addition to previous work on Gröbner bases in satisfiability checking [5,26], this result suggests that the use of our algebraic encoding, as well as Gröbner bases, brings benefits to Boolean logic and could be worthwhile to investigate further. Another interesting direction of further work could be the improvement of the Gröbner basis computation step. This could for example be done by studying the effects of different monomial orders or by implementing the Buchberger-Moeller algorithm [45] that can directly compute a Gröbner basis from a given set of zeros without first constructing a DNF.

Acknowledgements
We thank our anonymous reviewers and the SC^2 community for their valuable suggestions. Further, we thank Martin Bromberger and Christoph Weidenbach for proofreading.

Author contributions
All authors contributed to all sections of the paper.

Funding Information
Open Access funding enabled and organized by Projekt DEAL.

Data availability statement
Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

Declaration

Conflict of interest The authors declare that they have no conflict of interest.

Author details
[1]Department of Mathematics, Saarland University, Saarbrücken, Germany
[2]Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
[3]Graduate School of Computer Science, Saarbrücken, Germany.

Received: 29 February 2024 Revised: 19 December 2024 Accepted: 28 December 2024
Published online: 26 July 2025

References

1. Brayton, R.K., Hachtel, G.D., McMullen, C., Sangiovanni-Vincentelli, A.: Logic Minimization Algorithms for VLSI Synthesis. The Springer International Series in Engineering and Computer Science. Springer, New York (1984). https://doi.org/10.1007/978-1-4613-2821-6
2. Rudell, R.L.: Multiple-valued logic minimization for PLA synthesis. Technical report, EECS Department, University of California, Berkeley (1986). http://www2.eecs.berkeley.edu/Pubs/TechRpts/1986/734.html
3. Buchfuhrer, D., Umans, C.: The complexity of Boolean formula minimization. In: Automata, Languages and Programming, pp. 24–35. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70575-8_3
4. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: What's hard about Boolean functional synthesis? In: Computer Aided Verification, pp. 251–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_14
5. Condrat, C., Kalla, P.: A Gröbner basis approach to CNF-formulae preprocessing. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 618–631. Springer, Berlin, Heidelberg (2007).https://doi.org/10.1007/978-3-540-71209-1_48

6. Kreuzer, M., Robbiano, L.: Computational Commutative Algebra 1. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/978-3-540-70628-1
7. McCalpin, J.D.: Mapping addresses to L3/CHA slices in Intel processors. Technical report, TACC, The University of Texas at Austin (2021). https://doi.org/10.26153/tsw/14539
8. Peterson, W.W., Brown, D.T.: Cyclic codes for error detection. Proceedings of the IRE **49**(1), 228–235 (1961). https://doi.org/10.1109/JRPROC.1961.287814
9. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Theory and Applications of Satisfiability Testing - SAT 2009, pp. 244–257. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_24
10. Gerlach, L., Schwarz, S., Faroß, N., Schwarz, M.: Efficient and generic microarchitectural hash-function recovery. In: 2024 IEEE Symposium on Security and Privacy (SP), pp. 28–28. IEEE Computer Society, Los Alamitos (2024). https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00028
11. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic LTLf synthesis. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, pp. 1362–1369. IJCAI, California (2017). https://doi.org/10.24963/IJCAI.2017/189
12. Wang, L.-T., Chang, Y.-W., Cheng, K.-T.T.: Electronic Design Automation: Synthesis, Verification, and Test. Morgan Kaufmann, San Francisco (2009). https://dl.acm.org/doi/10.5555/2843514
13. The Sage Developers: Sage Mathematics Software, Version 9.0. http://www.sagemath.org (2023)
14. Decker, W., Greuel, G.-M., Pfister, G., Schönemann, H.: Singular — A computer algebra system for polynomial computations, Version 4.3.0. http://www.singular.uni-kl.de (2022)
15. GNU Linear Programming Kit, Version 4.32. http://www.gnu.org/software/glpk/glpk.html (2008)
16. Faroß, N., Schwarz, S.: Gröbner bases for Boolean function minimization. In: Proceedings of the 8th SC-Square Workshop, pp. 61–68. CEUR-WS.org, Aachen (2023). https://ceur-ws.org/Vol-3455/short4.pdf
17. Quine, W.V.: The problem of simplifying truth functions. The American mathematical monthly **59**(8), 521–531 (1952). https://doi.org/10.1080/00029890.1952.11988183
18. McCluskey, E.J.: Minimization of Boolean functions. The Bell System Technical Journal **35**(6), 1417–1444 (1956). https://doi.org/10.1002/j.1538-7305.1956.tb03835.x
19. Mishchenko, A., Perkowski, M.: Fast heuristic minimization of exclusive-sums-of-products. 5th International Reed-Muller Workshop (2001)
20. Brayton, R.K., Rudell, R., Sangiovanni-Vincentelli, A., Wang, A.R.: MIS: a multiple-level logic optimization system. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **6**(6), 1062–1081 (1987). https://doi.org/10.1109/TCAD.1987.1270347
21. Darringer, J.A., Brand, D., Gerbi, J.V., Joyner, W.H., Trevillyan, L.: LSS: A system for production logic synthesis. IBM Journal of Research and Development **28**(5), 537–545 (1984). https://doi.org/10.1147/rd.285.0537
22. Perkowski, M.: Multi-level logic minimization. http://web.cecs.pdx.edu/~mperkows/=FSM/finite-sm/node17.html (Accessed: 28.03.2023)
23. Soeken, M., Haaswijk, W., Testa, E., Mishchenko, A., Amarù, L.G., Brayton, R.K., Micheli, G.D.: Practical exact synthesis. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 309–314. IEEE, New York (2018). https://doi.org/10.23919/DATE.2018.8342027
24. Haaswijk, W., Soeken, M., Mishchenko, A., Micheli, G.D.: SAT-based exact synthesis: encodings, topology families, and parallelism. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(4), 871–884 (2020). https://doi.org/10.1109/TCAD.2019.2897703
25. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design. IEEE, New York (2013). https://doi.org/10.1109/FMCAD.2013.6679385
26. Nguyen, T.: Combinations of Boolean Gröbner bases and SAT solvers. PhD thesis, University of Kaiserslautern (2014). https://doi.org/10.13140/RG.2.2.25392.92167
27. Gao, S.: Counting zeros over finite fields with Gröbner bases. Master's thesis, Carnegie Mellon University (2009). https://www.cs.cmu.edu/%7Esicung/papers/MS_thesis.pdf
28. Ritirc, D., Biere, A., Kauers, M.: Improving and extending the algebraic approach for verifying gate-level multipliers. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1556–1561. IEEE, New York (2018). https://doi.org/10.23919/DATE.2018.8342263
29. Kaufmann, D., Biere, A., Kauers, M.: Verifying large multipliers by combining SAT and computer algebra. In: 2019 Formal Methods in Computer Aided Design (FMCAD), pp. 28–36. IEEE, New York (2019). https://doi.org/10.23919/FMCAD.2019.8894250
30. Kaufmann, D., Biere, A., Kauers, M.: Incremental column-wise verification of arithmetic circuits using computer algebra. Formal Methods in System Design **56**(1), 22–54 (2020). https://doi.org/10.1007/s10703-018-00329-2
31. Hader, T., Kaufmann, D., Kovacs, L.: SMT solving over finite field arithmetic. In: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, pp. 238–256. EasyChair, Manchester (2023). https://doi.org/10.29007/4n6w
32. Ozdemir, A., Kremer, G., Tinelli, C., Barrett, C.W.: Satisfiability modulo finite fields. In: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009). Chap. 26. https://doi.org/10.3233/978-1-58603-929-5-825
33. Junges, S., Loup, U., Corzilius, F., Ábrahám, E.: On Gröbner bases in the context of satisfiability-modulo-theories solving over the real numbers. In: Algebraic Informatics, pp. 186–198. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40663-8_18
34. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, New York (1972). https://doi.org/10.2307/2272104
35. Ghorpade, S.R.: A note on Nullstellensatz over finite fields. Contemporary Mathematics (2018)
36. Vazirani, V.: Approximation Algorithms. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-662-04565-7

37. Schwarz, S., Faroß, N.: Artifact: Gröbner Bases for Boolean Function Minimization. Zenodo (2024). https://doi.org/10.5281/zenodo.13935108
38. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Computer Aided Verification, pp. 24–40. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
39. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: a versatile and industrial-strength SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, vol. 13243, pp. 415–442. Springer, Berlin, Heidelberg (2022).https://doi.org/10.1007/978-3-030-99524-9_24
40. Padhi, S., Polgreen, E., Raghothaman, M., Reynolds, A., Udupa, A.: The SyGuS Language Standard Version 2.1. https://doi.org/10.48550/arXiv.2312.06001 (2023)
41. Hack, S.: Synthesis of Loop-free Programs. https://github.com/shack/synth (Accessed: 29.02.2023)
42. Wegener, I.: The Complexity of Boolean Functions. Wiley-Teubner, Stuttgart (1987). http://ls2-www.cs.uni-dortmund.de/monographs/bluebook/
43. Koopman, P., Chakravarty, T.: Cyclic redundancy code (CRC) polynomial selection for embedded networks. In: International Conference on Dependable Systems and Networks, pp. 145–154. IEEE, New York (2004). https://doi.org/10.1109/DSN.2004.1311885
44. Cook, G.: CRC RevEng. https://reveng.sourceforge.io/crc-catalogue/all.htm (Accessed: 29.02.2023)
45. Möller, H.M., Buchberger, B..: The construction of multivariate polynomials with preassigned zeros. In: Computer Algebra, pp. 24–31. Springer, Berlin, Heidelberg (1982). https://doi.org/10.1007/3-540-11607-9_3

Publisher's Note