REMaQE: Reverse Engineering Math Equations from Executables

MEET UDESHI and PRASHANTH KRISHNAMURTHY, ECE, NYU Tandon School of Engineering, Brooklyn, NY, USA
HAMMOND PEARCE, School of Computer Science and Engineering, UNSW, Sydney, Australia
RAMESH KARRI and FARSHAD KHORRAMI, ECE, NYU Tandon School of Engineering, Brooklyn, NY, USA

Cybersecurity attacks on embedded devices for industrial control systems and cyber-physical systems may cause catastrophic physical damage as well as economic loss. This could be achieved by infecting device binaries with malware that modifies the physical characteristics of the system operation. Mitigating such attacks benefits from reverse engineering tools that recover sufficient semantic knowledge in terms of mathematical equations of the implemented algorithm. Conventional reverse engineering tools can decompile binaries to low-level code, but offer little semantic insight. This article proposes the REMaQE automated framework for reverse engineering of math equations from binary executables. Improving over state-of-the-art, REMaQE handles equation parameters accessed via registers, the stack, global memory, or pointers, and can reverse engineer equations from object-oriented implementations such as C++ classes. Using REMaQE, we discovered a bug in the Linux kernel thermal monitoring tool "tmon." To evaluate REMaQE, we generate a dataset of 25,096 binaries with math equations implemented in C and Simulink. REMaQE successfully recovers a semantically matching equation for all 25,096 binaries. REMaQE executes in 0.48 seconds on average and in up to 2 seconds for complex equations. Real-time execution enables integration in an interactive math-oriented reverse engineering workflow.

CCS Concepts: • **Security and privacy → Embedded systems security**; **Software reverse engineering**;

Additional Key Words and Phrases: binary reverse engineering, embedded systems, symbolic execution, mathematical equations

ACM Reference format:
Meet Udeshi, Prashanth Krishnamurthy, Hammond Pearce, Ramesh Karri, and Farshad Khorrami. 2024. REMaQE: Reverse Engineering Math Equations from Executables. *ACM Trans. Cyber-Phys. Syst.* 8, 4, Article 43 (November 2024), 25 pages.
https://doi.org/10.1145/3699674

This work was supported in part by the Sponsor Office of Naval Research under the Grant #N00014-22-1-2153 and the Sponsor National Science Foundation under the Grant #2039615.
Authors' Contact Information: Meet Udeshi (corresponding author), ECE, NYU Tandon School of Engineering, Brooklyn, NY, USA; e-mail: m.udeshi@nyu.edu; Prashanth Krishnamurthy, ECE, NYU Tandon School of Engineering, Brooklyn, NY, USA; e-mail: prashanth.krishnamurthy@nyu.edu; Hammond Pearce, School of Computer Science and Engineering, UNSW, Sydney, Australia; e-mail: hammond.pearce@unsw.edu.au; Ramesh Karri, ECE, NYU Tandon School of Engineering, Brooklyn, NY, USA; e-mail: rkarri@nyu.edu; Farshad Khorrami, ECE, NYU Tandon School of Engineering, Brooklyn, NY, USA; e-mail: khorrami@nyu.edu.

ACM 2378-9638/2024/11-ART43
https://doi.org/10.1145/3699674

1 Introduction

Embedded systems in **industrial control systems (ICS)** and **cyber-physical systems (CPS)** are vulnerable to process-aware cyber-attacks that compromise the physical processes under control [15, 17, 37]. Such attacks can breach safety-critical requirements, causing real-world destruction and harm. For example, Stuxnet injected malware in the **programmable logic controllers (PLC)** of a nuclear facility to adversely modify the centrifuge motor frequencies [20]. The BlackEnergy attack against Ukraine's power grid opened several breakers at once to cause a power outage [21]. Covert attacks are also possible: Krishnamurthy et al. [18] develop a stealthy communication channel that uses analog emissions from a CPS. This method maintains the closed-loop process characteristics by factoring in controller dynamics during attack design. Civilian **unmanned aerial vehicles (UAVs)** (drones) are being targetted to inject malicious software trojans in the controller and allow attackers to gain control of the drone [1]. Sun et al. [37] present a survey of attacks on ICS, highlighting more examples of control logic modification attacks that impact PLCs.

One can analyze the cybersecurity of embedded systems by leveraging semantic understanding of the physical process features, such as mathematical models, control algorithms, and dynamic behavior of the process. For example, Yang et al. [40] derive control invariants of physical processes using PLC runtime logs and detect changes in these invariants to signal an intrusion. Badenhop et al. [4] reverse engineer the proprietary Z-Wave transceiver using static and dynamic analysis to verify communication security properties. The Trusted Safety Verifier in [26] is implemented as a bump-in-the-wire and verifies safety-critical code for PLCs by recovering a semantic graph of the program and asserting safety-critical properties using model checking. Bourbouh et al. [5] implement a reverse compilation framework from Lustre (low-level synchronous data-flow language) to Simulink (high-level modelling framework) to add semantic context to the Simulink models.

On the offensive side, McLaughlin [25] shows that recovering boolean expressions of the PLC control code can automatically generate dynamic malware payloads. The CLIK attack on PLCs [13] decompiles the control logic to high-level automation languages, making malicious code changes to disrupt the physical process. The Laddis decompiler [32] enables the attacker to mount a denial of engineering operations attack on a PLC by modifying the ladder logic.

While there are many methods to reverse engineer embedded systems and extract semantic knowledge for cybersecurity, this article focuses on the analysis and reverse engineering of binary executables of the embedded devices (e.g., PLCs) which interact with physical sensors and actuators. Reverse engineering the mathematical models and control algorithms implemented in the binaries can reveal the semantic knowledge necessary to understand these embedded systems. This semantic knowledge is useful for applications such as (a) analyzing malicious changes injected in the binary by malware, (b) recovering details of legacy systems without source code, (c) examining adversarial systems, such as UAVs or drones used for reconnaissance, (d) identifying vulnerabilities in implemented systems for patching or exploitation, and (e) debugging during implementation of mathematical algorithms.

While binary compilation works well in the forward direction (math equation → binary executable), reversing this process is difficult. Math equations compiled into binaries are deployed on a diverse range of embedded hardware platforms and target a variety of processor architectures. The implementation of math equations involves platform-specific details which offer no semantic information. Disassembly and decompilation tools like IDA Pro [11] and Ghidra [28] are useful for analysis of binaries, but they do not recover semantic information. Symbolic execution is useful to gain semantic insight into a binary with dynamic analysis [34], but the recovered semantic information is not presented as human-friendly math equations. There is thus a need for a framework

that can automatically reverse engineer math equations from their binaries and present them in a human-friendly form. To this end, we propose **Reverse Engineering Math Equations from Executables (REMaQE)**.

1.1 Contributions

The REMaQE framework implements automatic parameter analysis and algebraic simplification methods to automate the reverse engineering of mathematical equations from binary executables. REMaQE is built on top of existing binary analysis frameworks and symbolic mathematics tools. REMaQE employs automatic parameter analysis of functions to identify important metadata regarding the function arguments stored in register, stack, global memory, or accessed via pointer. According to the best of our knowledge, REMaQE is the *first* work able to reverse engineer math equations from a variety of function implementations, including object-oriented C++ code. Simplification of math equations in REMaQE is performed via math-aware algebraic methods. This overcomes limitations of other approaches such as machine learning methods for equation simplification and enables REMaQE to simplify complex conditional equations. Unlike state-of-the-art methods that look for patterns of well-known algorithms, REMaQE refines the extracted semantic information and displays it as math equations, enabling general-purpose applications on real-world use cases. REMaQE can be used in conjunction with existing reverse engineering tools to provide math-oriented semantic information in a reverse engineering workflow. Parameter analysis and algebraic simplification sets REMaQE apart from prior semantic reverse engineering approaches. Section 3 discusses the strengths of REMaQE over existing works. The contributions of this article are as follows:

(1) The REMaQE framework for reverse engineering mathematical equations from binary executables, which offers two major improvements over existing approaches:
   (a) Automatic parameter analysis to recognize input, output, and constant parameters in an implemented equation. This enables reverse engineering of object-oriented implementations, such as C++ classes and struct pointer based C functions.
   (b) Algebraic simplification to transform extracted symbolic expressions into easily understandable math equations. This handles much more complex expressions compared to existing approaches that use machine learning methods.
(2) A dataset of 25,096 compiled binary executables and their corresponding 3,137 math equations with their implementation as C code and Simulink models. This is suitable for evaluating reverse-engineering tools and is made available at [38].

The article is organized as follows: Section 2 offers a motivating example of the Linux "tmon" controller bug found by REMaQE, Section 3 reviews related work, Section 4 details REMaQE's implementation, Section 5 explains the evaluation procedure and dataset generation methodology, Section 6 reports results, Sections 7 and 8 showcase two case studies on reverse engineering the ArduPilot [3] auto-pilot C++ firmware and an OpenPLC [2] **Proportional-Integral-Derivative (PID)** controller using REMaQE, and Section 9 concludes and explores future work.

2 Motivation—Linux Kernel PID Controller

This section presents the reverse engineering of the Linux kernel thermal monitoring tool "tmon"[1] that uses a PID controller. We demonstrate how the math equations recovered by REMaQE help in uncovering a bug in the controller's implementation. Figure 1(a) shows the source code of the

[1] https://github.com/torvalds/linux/blob/v6.3/tools/thermal/tmon/pid.c

```c
void controller_handler(const double xk, double *
     yk) {
  double ek;
  double p_term, i_term, d_term;
  ek = p_param.t_target - xk; /* error */
  if (ek >= 3.0) {
    syslog(LOG_DEBUG, "PID:␣%3.1f␣Below␣set␣point
         ␣%3.1f,␣stop\n", xk, p_param.t_target);
    controller_reset();
    *yk = 0.0;
    return;
  }
  /* compute intermediate PID terms */
  p_term = -p_param.kp * (xk - xk_1);
  i_term = p_param.kp * p_param.ki
           * p_param.ts * ek;
  d_term = -p_param.kp * p_param.kd
           * (xk - 2 * xk_1 + xk_2) / p_param.ts;
  /* compute output */
  *yk += p_term + i_term + d_term;
  /* update sample data */
  xk_1 = xk;
  xk_2 = xk_1;
  /* clamp output adjustment range */
  if (*yk < -LIMIT_HIGH)
    *yk = -LIMIT_HIGH;
  else if (*yk > -LIMIT_LOW)
    *yk = -LIMIT_LOW;
  p_param.y_k = *yk;
  set_ctrl_state(lround(fabs(p_param.y_k)));
}
```

(a) C source of `controller_handler` in "tmon". The blue and yellow highlighted lines indicate the implementation bugs.

```c
void FUN_00015194(double *param_1,undefined4 param_2
     ) {
  double in_d0, dVar1, dVar2;
  if (3.0 <= _DAT_00018448 - in_d0) {
    syslog(7,"PID:␣%3.1f␣Below␣set␣point␣%3.1f,␣stop
         \n");
    FUN_00015150();
    *(undefined4 *)param_1 = 0;
    *(undefined4 *)((int)param_1 + 4) = 0;
    return;
  }
  dVar2 = (((in_d0 - (_DAT_00018458 + _DAT_00018458)
       ) + _DAT_00018460) * -(_DAT_00018420 *
       _DAT_00018430)) / DAT_00018438 + -(
       _DAT_00018420 * (in_d0 - _DAT_00018458)) +
       _DAT_00018420 * _DAT_00018428 * DAT_00018438
       * (_DAT_00018448 - in_d0) + *param_1;
  dVar1 = -95.0;
  *param_1 = dVar2;
  _DAT_00018458 = in_d0;
  _DAT_00018460 = in_d0;
  if (-95.0 <= dVar2) {
    dVar1 = -2.0;
    if (dVar2 < -2.0) goto LAB_00015270;
  }
  *param_1 = dVar1;
LAB_00015270:
  _DAT_00018450 = *param_1;
  lround((double)CONCAT44(param_2,param_1));
  FUN_00014ddc();
  return;
}
```

(b) `controller_handler` decompiled with Ghidra.

```
<FP64 fpToFP((if reg_r0_4_32{UNINITIALIZED} == 0xffffffff && reg_r0_4_32{UNINITIALIZED} + 0x4 == 0x3 &&
     (1 & ~(<...>[0:0] & 1 ^ <...>[0:0] & 1)) == 1 then reg_d0_1_64{UNINITIALIZED} else (if ((LShR
     (<...>, <...>)[0:0] | <...>[0:0] ^ <...>[0:0]) & 1) == 1 && reg_r0_4_32{UNINITIALIZED} == 0
     xfffffff0 && ((LShR(<...>, <...>)[0:0] ^ 1) & 1) != 0 && (1 & ~(<...> & <...> ^ <...> & <...>)) !=
      1 then fpToIEEEBV(fpAbs(fpAdd(RM.RM_NearestTiesEven, fpAdd(RM.RM_NearestTiesEven, fpDiv(RM.
     RM_NearestTiesEven, fpMul(RM.RM_NearestTiesEven, <...>, <...>), FPV(0.0, DOUBLE)), fpAdd(RM.
     RM_NearestTiesEven, fpNeg(<...>), fpMul(RM.RM_NearestTiesEven, <...>, <...>))), fpToFP(
     mem_fffffff0_5_64{UNINITIALIZED}, DOUBLE)))) else 0x0)), DOUBLE)>
```

(c) Symbolic expression generated with Angr.

$$t = x_8 - \frac{x_1 x_3}{x_4}(x_0 - 2x_6 + x_7) - x_1 x_2 x_4 (x_0 - x_5) - x_1 (x_0 - x_6)$$

$$y_2 = \begin{cases} t & \text{for } x_5 - x_0 \le k_2 \text{ and } k_4 \ge t \text{ and } k_3 \le t \\ k_3 & \text{for } x_5 - x_0 \le k_2 \text{ and } k_3 > t \\ k_4 & \text{for } x_5 - x_0 \le k_2 \text{ and } k_3 \le t \text{ and } k_4 < t \\ 0 & \text{otherwise} \end{cases}$$

$$y_3 = \begin{cases} x_0 & \text{for } x_5 - x_0 \le k_2 \\ 0 & \text{otherwise} \end{cases}$$

$$y_4 = \begin{cases} x_0 & \text{for } x_5 - x_0 \le k_2 \\ 0 & \text{otherwise} \end{cases}$$

(d) Math equations recovered by REMaQE. The term t is introduced to display the y_2 equation clearly.

Fig. 1. Reverse engineering the Linux "tmon" thermal controller with different tools: (a) C source code, (b) decompilation with Ghidra, (c) symbolic execution with Angr, (d) math equations recovered with REMaQE.

`controller_handler` function that implements the PID controller. The source code is provided only for discussion and is not utilized during reverse engineering. A pre-compiled binary for the **ARM 32-bit Hard-Float (ARM32-HF)** target is taken from the package repository of the Alpine Linux distribution,[2] which is popularly used on embedded platforms like Raspberry Pi. For a comparison, the generated binary is reverse engineered using different tools namely, Ghidra [28] for decompilation, Angr [34] for symbolic execution, and REMaQE to recover math equations.

[2] https://dl-cdn.alpinelinux.org/alpine/v3.18/community/armhf/linux-tools-tmon-6.3.12-r0.apk

The function is decompiled using Ghidra to generate the C source shown in Figure 1(b). The decompiled `controller_handler` function is presented in a complicated form by Ghidra—it is cluttered with global variable access, struct pointer dereferencing and `goto` statements for control flow. This makes it difficult to understand what is implemented, even with the knowledge that the function contains a PID controller. Ghidra has generated a faithful representation. However, these implementation details present in the syntax are obscuring the semantic information of this function. Thus, significant effort is required to clean up and extract a math equation of the PID controller.

Note that the function name is not recovered as the pre-compiled binary is stripped of symbols. The user needs to identify the function of interest to analyze by looking at either the disassembly or the decompiled output of Ghidra. In this manner, REMaQE provides additional semantic information for an interactive reverse engineering workflow. Figure 1(c) shows the symbolic expression generated by Angr. Even though the generated expression has recovered sufficient semantic information about the source function, the implemented equations are not readily apparent to the human analyst.

Figure 1(d) shows the math equations recovered by REMaQE for the various output parameters of the function. REMaQE can not recover variable names from the stripped binary; hence it names all inputs, outputs, and constants as x_n, y_n, and k_n, respectively. For this example, REMaQE has identified the function parameters *yk, xk_1, xk_2 as inputs x_8, x_6, x_7 and outputs y_2, y_3, y_4, respectively. These three variables are accessed in different ways (pointer dereference, global access) and they have been used as both inputs and outputs in the function; yet, REMaQE has uniformly presented the output equations clearly. This is discussed in detail in Section 4.3.

From the equations recovered by REMaQE, we see that y_3 and y_4 will take the same value after the function is run once. When the function is called again, variables xk_1 and xk_2, now used as inputs x_6 and x_7, will be the same. In the PID control equation, the D-term $\frac{x_1 x_3}{x_4}(x_0 - 2x_6 + x_7)$, will simplify to $\frac{x_1 x_3}{x_4}(x_0 - x_6)$. This degrades the three-point approximation of the D-term to a two-point one, impacting the quality of the PID controller. The reason is clear from the yellow highlighted lines in Figure 1(a): line 20 wrongly reassigns xk_1 = xk before line 21 assigns xk_2 = xk_1, which means that both variables take the value xk. The correct implementation would be to switch the order of assignment and assign xk_2 before reassigning xk_1. This can be introduced due to either human error or malware which swaps the order of a few assembly instructions. The impact on the code and the execution is insignificant, yet the physical characteristics of the controller change noticeably and may have real-world consequences. We have submitted a patch to fix this bug.[3] This example demonstrates the advantage of reverse engineering using REMaQE by providing a semantically rich understanding of the implemented math equations for binary analysis.

3 Related Work

As discussed in Sections 1 and 2, disassembly and decompilation techniques are unable to provide relevant semantic information of a program, while REMaQE is able to portray the program as semantically equivalent math equations. The goal of decompilation is to reconstruct the original source code from binary executables as accurately as possible. Decompilers intend to recover many implementation details of the program such as variable types, memory layout, control flow, and data flow [8]. These syntactical details are necessary when the recovered programs need to be

[3] https://patchwork.kernel.org/project/linux-pm/patch/20230822184940.31316-1-mudeshi1209@gmail.com/

represented in high-level source languages like C or Java. However, due to the noisy and obfuscating nature of compilation, decompilation is an indeterminate process; hence, the recovered code may obscure the original high-level semantic meaning. The reverse engineering approach of REMaQE aims to refine and discard these implementation details before generating human-friendly equations using conventional math symbols and operations. REMaQE is not intended to replace decompilers or decompilation techniques. Instead, it can be integrated with the user interface of decompilation tools to offer the semantic information of equations alongside useful information extracted by the decompiler, for example, the C code of a function along with the math equation.

Manually recovering math equations from the binaries or obscured decompilation outputs requires subject matter expertise of identifying and clustering code sequences, mapping them to math primitives, and understanding how identified blocks will combine and simplify. Automated approaches tailored to the embedded systems can use knowledge about the compilation tool chain and build a better representation in domain-specific languages. For example, executables from control devices can be reverse engineered to automation languages like Ladder Logic and Instruction List [13, 22, 30, 32]. These methods offer better abstraction than languages like assembly or C, but they too cannot represent the semantic information. This necessitates domain expertise and manual effort to recover math equations.

Programs may be intentionally obfuscated. For example, malware is obfuscated to hide its intent and avoid detection or analysis [29]. Jha et al. [12] use oracle-guided program synthesis to develop a semantic understanding to deobfuscate malware. Malware obfuscation uses bit-manipulation operations and deliberate complex control flows specifically to hinder analysis and prevent reverse engineering. Deobfuscation is outside the scope of the current article (see Section 4.6).

Symbolic execution is a technique to gain semantic insight into a binary with dynamic analysis. Symbolic execution explores all execution paths through the program and captures symbolic expressions [34]. Primary use cases are automatic test generation [6, 35], exploit detection and generation [7, 33], and reverse engineering [10, 26]. The symbolic expressions are "solved" using **satisfiability modulo theory (SMT)** solvers to identify concrete inputs. Although symbolic expressions are generated to assist SMT solvers, they are not presented as human-friendly equations. They contain platform-specific implementation details that obscure the semantic information (see Section 4.5). Simply printing the symbolic expressions in a human-readable format is not sufficient. REMaQE employs algebraic simplification techniques to clean up the expressions.

Recent approaches to extract high-level semantic information [14, 16, 36] have used domain knowledge and assumptions to match the extracted information to patterns of well-known algorithms. MISMO [36] employs a template of popular control algorithms and performs semantic matching on expressions extracted using symbolic execution to determine which control algorithm is implemented. DisPatch [16] targets controllers for robotic aerial vehicles, and identifies customized implementations of the PID controller.

MISMO authors have analyzed the "tmon" tool and identified that the implemented PID control equations do not match the expected pattern. In Figure 1(a), the blue highlighted term `p_param.kp` is multiplied for the I and D terms, due to which this implementation fails to match any of their template patterns. Assuming availability of source code, further manual inspection reveals the implementation bug. Using REMaQE, the user can identify this mismatch, as highlighted by the blue terms x_1 in the equations in Figure 1(d). However, MISMO's pattern matching *fails* to highlight the wrong assignment to `xk_1` and `xk_2` that causes the second implementation bug described in Section 2. Furthermore, unlike MISMO, REMaQE's reverse engineering output enables these bugs to be detected without access to source code. Prior methods are domain-specific and can be sensitive to alterations in implementation that breach domain-specific assumptions. Although they extract semantic information similar to REMaQE, implementing and deploying such techniques

Table 1. Feature Comparison of REMaQE with Existing Approaches for Reverse Engineering Math Equations

		DisPatch [16]	MISMO [36]	PERFUME [39]	REMaQE (our)
Fixed patterns		●	●	●	●
General purpose		○	○	●	●
Parameter Access	register	◐	◐	●	●
	stack	◐	◐	○	●
	pointer	◐	◐	○	●
	global	◐	◐	○	●
Conditionals		○	◐	○	●
Simplification		○	○	◐	●

●, fully supported; ◐, supported for some cases; ○, not supported.

necessitates combining domain expert knowledge with reverse engineering. These methods may fail on outlier cases. In contrast, REMaQE provides a versatile approach based on a fundamental set of assumptions that can be readily expanded, enabling its application to various domains with minimal adaptation.

PERFUME [39] overcomes limitations in prior domain-specific approaches by following a generic approach which does not rely on semantic pattern matching. PERFUME is able to present the math equation pertaining to the implemented algorithm in a human-friendly form. Among existing works, PERFUME is most similar to REMaQE. PERFUME uses symbolic execution to extract the semantic information in the form of symbolic expressions, and trains a machine learning based sequence-to-sequence machine translation model to "translate" that symbolic expression into a simplified equation. The framework is offered as a plugin for Ghidra that augments decompiled output with semantic information and integrates into an interactive reverse engineering workflow.

PERFUME, however, cannot analyze functions with parameters accessed through the stack, global memory, or pointers. Hence, PERFUME fails to analyze the "tmon" tool in Section 2, because this function uses a struct, a pointer dereference, and global variables to read inputs and write outputs. In contrast, REMaQE employs a parameter analysis pass collecting metadata about each parameter's storage location. REMaQE can reverse engineer a wider variety of functions, including those found in object-oriented implementations in real-world programs like "tmon." Additionally, while PERFUME's machine translation model struggles to simplify long and complex symbolic expressions arising from conditional logic, REMaQE's algebraic simplification stage adopts a math-aware algorithmic approach. It handles complex expressions generated from functions with floating-point comparisons and conditional logic. This deterministic simplification scales based on the complexity of the simplified expression, not the input expression. PERFUME's machine translation model cannot simplify the complex conditional expression (Figure 1(c)) generated for the output clamping logic of the "tmon" controller (Figure 1(a), lines 23–26), whereas REMaQE produces a compact, human-friendly equation. REMaQE's parameter analysis and algebraic simplification features set it apart from PERFUME and other symbolic execution approaches.

Table 1 compares the features of existing approaches and REMaQE. DisPatch and MISMO require pre-programmed patterns of well-known control algorithms and extract information pertaining to

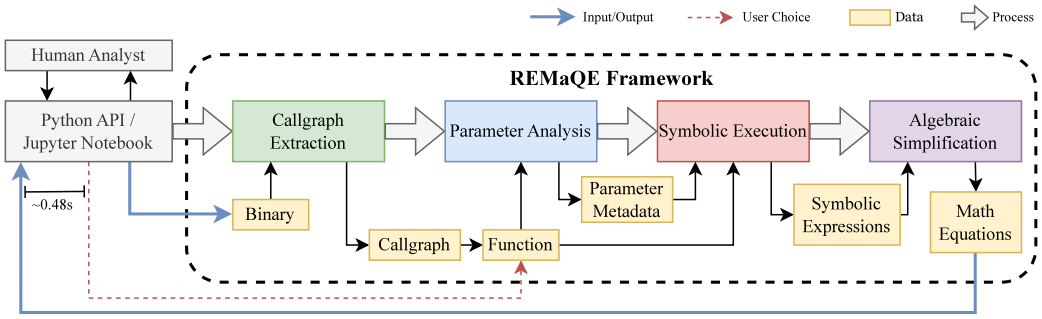

Fig. 2. The REMaQE framework. Average execution time of the pipeline is 0.48 seconds, from when the user provides which function to reverse engineer, to when REMaQE returns the math equations.

the matched pattern. They support all kinds of parameter access and conditional statements, but as part of the matched pattern. On the other hand, PERFUME is general-purpose, however it can only handle parameters present in registers, severely restricting its scope. REMaQE is general-purpose, yet handles all kinds of parameter access. It also handles conditional logic in a general context.

4 Implementing REMaQE

4.1 Background on Symbolic Execution

Symbolic execution involves running a program in an emulated environment using symbols instead of concrete values. This process generates a symbolic **expression tree (ET)**, a type of abstract syntax tree. When conditional branches depend on symbolic data, the execution forks to follow both taken and not taken paths, adding a constraint or predicate to each path based on the branch condition. Constraints accumulate as the execution progresses through each branch. Upon reaching an equivalence point, such as a function return statement, the ETs can be combined into one, utilizing the constraints gathered along each path. For example, symbolic execution of the binary code of `controller_handler` shown in Figure 1(a) forks paths on the branch instruction on line 23, leading to two execution paths, one which takes the branch and the other which does not. The paths are merged into one at the function return, and the ET for the return value is simplified by REMaQE into a piecewise form as shown in Figure 1(d).

When reverse engineering math equations, symbolic execution offers two big advantages. First, operations representing the math equation are naturally captured and tracked. Second, the forking on branches ensures that all code is explored during execution, yielding a full picture of the implemented math equation. Code coverage is important, as failing to explore the entire code base can yield a math equation that is undefined for certain input values. REMaQE relies on the Angr binary analysis framework [34] for dynamic analysis and symbolic execution.

4.2 Overview of REMaQE

Figure 2 presents an overview of the REMaQE framework. REMaQE focuses on reverse engineering math equations of one function at a time. This allows reverse engineering to proceed in a modular fashion, similar to the approach a human analyst may follow manually. REMaQE first extracts a call graph of the functions present in the executable. The user then selects a function for REMaQE to analyze. The function runs through the parameter analysis stage to gather information about input, output and constant parameters that express the math equation. This information is packaged into the parameter metadata. The symbolic execution stage then runs the function with initialized

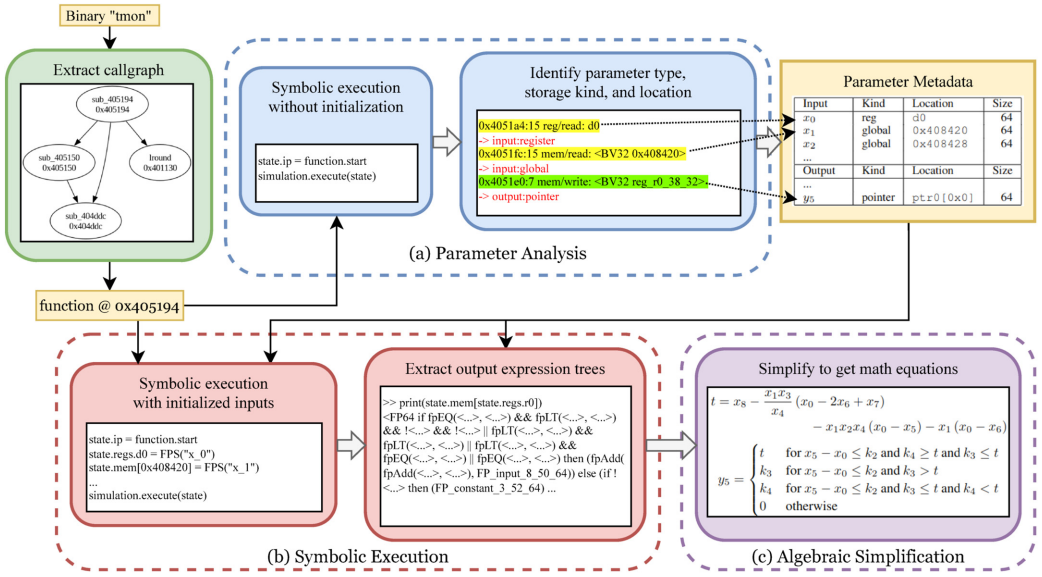

Fig. 3. REMaQE reverse engineering pipeline. The `controller_handler` function from Figure 1 is used as an example, and intermediate outputs are shown for each stage through the pipeline: (a) Parameter analysis automatically identifies the input, output and constant parameters of the function along with their kind and storage location, (b) Symbolic execution runs the function with properly initialized symbolic inputs and gathers the output symbolic ETs, and (c) Algebraic simplification converts the output ET to a human-friendly math equation.

inputs and extracts the symbolic ETs for each output. The output ETs are simplified by the algebraic simplification stage to generate the math equation.

REMaQE's function-by-function approach enables control over whether to represent function calls as-is or substitute them in the recovered equation. For instance, trigonometric functions might be implemented using approximating polynomials or platform-specific hardware extensions in libraries. However, such implementation details do not provide valuable information and may even cloud the equation's meaning. In such cases, it is better to represent these as function calls in the generated equation. Conversely, in some situations, it could be beneficial to substitute a called function's equation into the caller function's equation. The `controller_reset` function in "tmon" is one such example, where it is beneficial to be substituted. Certain functions which do not impact the output can be ignored, such as logging, printing, and error functions (e.g., `syslog`).

REMaQE provides an API in Python, which allows the user to analyze the executable function by function and to control each stage of REMaQE. The API can be used in Python scripts or in a Python command line interface like I-Python or Jupyter for interactive reverse engineering.

Figure 3 shows the REMaQE pipeline. The Linux "tmon" example discussed in Section 2 is used to explain the REMaQE pipeline. The `controller_handler` function is selected for analysis. Snippets of intermediate outputs of each stage are elaborated in Sections 4.3, 4.4, and 4.5.

4.3 Parameter Analysis

To invoke symbolic execution of a function, it is important that the execution entry state is initialized with symbolic parameters placed in the expected locations. To read the output ET, it is necessary to know where the function places the outputs. A math equation may use certain constants, which

Algorithm 1: Parameter Analysis

```
1:  procedure AnalyzeParams(F: function)
2:  T: execution trace ← SymbolicExecute(F)
3:  Inputs, Outputs, Constants ← empty lists
4:  for A: action in T do
5:      L ← StorageLocation(A)
6:      Reads, Writes ← GetReadsWrites(L)
7:      if Writes is empty and L is initialized then
8:          V ← GetValue(L)
9:          Append (L, V) to Constants
10:     else if first(Reads) < first(Writes) then
11:         Append L to Inputs
12:     end if
13:     if Writes is not empty then
14:         Append L to Outputs
15:     end if
16: end for
17: return Inputs, Outputs, Constants
18: end procedure
```

the tool should distinguish from inputs and extract their hard-coded value. The parameter analysis stage collects the information of the storage location of parameters.

The parameters of a function are defined as the inputs, outputs and constants which are used to express the implemented math equation, and pointers which hold the storage address of the parameters. Temporary variables which may be used in the function implementation are not considered as parameters. In the "tmon" example, the `controller_handler` function takes a pointer to `yk` as an argument. This pointer is not considered an input/output parameter by REMaQE, instead it is separately marked and the dereferenced value is considered as input/output. Similarly, the global variables `xk_1` and `xk_2` are considered as output parameters as they are written to by the function, even though they are not part of the function's return value.

The memory layout of data at the pointer address or inside a struct is not important since each separate value, whether it is dereferenced or accessed as a member, is considered a distinct parameter. Only the offset is necessary. This distinction helps to abstract different implementation details, allowing for a clean math equation recovery. The metadata for each parameter contains the following information: (i) Parameter name, (ii) Storage kind, and (iii) Storage location. Storage location, depending on kind, is either the architectural name of *register*, *stack* offset, *global* address, or *pointer* address and offset. The pointer parameters also require a storage location, since a concrete address pointing to memory containing symbolic parameters must be passed as an argument to the function. Constants may also be present as hard-coded immediate values in the instruction. This is indicated as the *immediate* kind, with the instruction address as location.

Algorithm 1 describes parameter analysis of a function. The parameter analysis pass has no knowledge of function parameters, so it does not initialize the symbolic execution state and relies on Angr to fill in uninitialized values when accessed. The "SymbolicExecute" call performs this uninitialized execution and records an execution trace. The execution trace is parsed and read/write accesses to each storage location are recorded in a sequential history. "GetReadsWrites" gathers this access history for each storage location. Locations with a first access as read are marked as inputs. These input locations are filtered and labeled as constant if they have no write access and contain

Table 2. Parameter Metadata Generated for the "tmon" PID Controller in Figure 1

Input	Kind	Location	Size
x_0	register	`d0`	64
x_1	global	`0x408420`	64
x_2	global	`0x408428`	64
x_3	global	`0x408430`	64
x_4	global	`0x408438`	64
x_5	global	`0x408448`	64
x_6	global	`0x408458`	64
x_7	global	`0x408460`	64
x_8	pointer	`ptr0[0x0]`	64
Pointer	**Kind**	**Location**	**Size**
`ptr0`	register	`r0`	32

Output	Kind	Location	Size	
y_0	register	`r0`	32	
y_1	register	`d0`	64	
y_2	global	`0x408450`	64	
y_3	global	`0x408458`	64	
y_4	global	`0x408460`	64	
y_5	pointer	`ptr0[0x0]`	64	
Constant	**Kind**	**Location**	**Size**	**Value**
k_2	global	`0x405298`	64	3
k_3	global	`0x4052a0`	64	−95
k_4	global	`0x4052a8`	64	−2

The input, output, constant and pointer parameters are listed, along with storage kind and location.

an initialized value. Uninitialized locations are filled with a symbolic value by Angr, differentiating inputs from constants since constant parameters are hard-coded in the binary. Some constants that are initialized at runtime (e.g., C++ class member assigned in the the class constructor) will be treated as inputs because symbolic execution begins at the entry point of the selected function, and initializations outside the function cannot be tracked. This does not affect the recovered equation except that the constant is treated as input and it's value cannot be extracted. Locations with at least one write access are marked as output. A single location can be both input and output, as the implementation may reuse storage locations.

Table 2 shows the parameter metadata obtained for the "tmon" `controller_handler` function. Input x_0 is assigned to the ARM 64-bit floating point register `d0`. Inputs x_1 to x_7 are loaded from global memory. Input x_8 is dereferenced from the pointer `ptr0` with offset 0. As indicated by the pointer table, `ptr0` is present in 32-bit integer register `r0`. Of the 6 outputs, 5 have the same location as inputs, yet they are distinguished as separate parameters of the equations. During the subsequent symbolic execution stage, register `d0` is initialized as the symbolic input x_0 at start and the ET for y_4 is read from it at the end. Similarly, the outputs y_0, y_2, y_3 are co-located with inputs x_8, x_6, x_7 and are handled accordingly.

4.4 Symbolic Execution

This stage initializes the symbolic state with appropriate symbols for each parameter, using the metadata generated during parameter analysis. The execution runs until the function returns, at which point all execution paths are merged into a final state. The parameter metadata is used to read the output ET from the correct storage locations in the final state. Due to the proper initialization in this stage, the output ET depends on the appropriate symbols, hence a math equation can be derived for the output. This leads to a subtle difference between the ET in Figure 3(b) and the Angr output in Figure 1(c), where the Angr output does not contain properly initialized symbols and hence cannot be directly simplified to a math equation, demonstrating the need for parameter analysis. Constants are treated as symbols during reverse engineering and are optionally substituted in the final equation, so that the constant values are not changed by any computation during execution or simplification. REMaQE recovers an accurate depiction of the constants which are hard-coded in the binary and how they are expressed in the math equation, including modifications during compilation like floating point approximation.

Algorithm 2: Simplify Conditional ETs

```
1: procedure SimplifyConditional(ET)
2: for all comparison C in ET do
3:     LHS, RHS ← CanonicalOrder(C)
4:     LT ← LHS < RHS
5:     EQ ← LHS == RHS
6:     BE ← BoolExp(C, LT, EQ)
7:     Replace C with BE in ET
8: end for
9: S ← QuineMcCluskey(ET)
10: for all boolean B in S do
11:     C ← GetComparison(B)
12:     Replace B with C in S
13: end for
14: return S
15: end procedure
```

4.5 Algebraic Simplification

The equations generated by reverse engineering have operations which follow the sequence of computations performed by the function's binary implementation. This representation may be cluttered with implementation-specific operations that need to be cleaned up and simplified to produced human-friendly equations. Conditional branches also generate complex expressions with deeply nested `if-then-else` statements. Such expressions can result when merging execution paths with constraints as described in Section 4.1. Figure 3(b) shows the internal representation of the ET obtained after symbolic execution of `controller_handler`. The complex expression is generated because of the conditional statements executed for the clamping operation in lines 23–26. Even though the clamping requires only two conditions, the generated ET represents the sequence of operations performed during symbolic execution which semantically represent the conditions in the function. A similar complex ET example can be seen in Figure B1 in Appendix B.

The sequence of operations is determined by two factors. First, when the C function is compiled to a target machine code, the conditional branches can be represented using a variety of instruction sequences as decided by the compiler. Second, Angr maps every instruction in the binary from the target's machine code to an intermediate representation required by it's symbolic execution engine. This further modifies the sequence of operations from what the binary's instructions describe. The only guarantee is that these modifications and mappings are semantically correct and represent the original equation. When the condition involves a comparison, a separate instruction performs the comparison and updates bits in a flag register. Later, conditional instructions check these bits of the flag register. When these instructions are mapped to the IR, Angr can generate bit-wise operations to perform checks, so these operations end up in the final expression.

REMaQE uses algebraic simplification to clean up implementation-specific operations and represent them as a simplified equation. The if-then-else operation of Angr requires a condition, a true clause, and a false clause. The condition ET must yield a boolean value. The conditions in math equations generally consist of one or more comparisons combined with boolean operations. Algebraic simplification exploits this to streamline the condition ET to simple comparisons before producing the equation.

Each condition ET in the full expression is simplified by Algorithm 2. The ETs generated by Angr are immutable and hashable to maintain consistency across multiple symbolic expressions of one execution. The "CanonicalOrder" function extracts the left and right ET of the comparison C, and orders them according to their hash value. Two booleans, LT and EQ, are generated to represent less-than and equal-to comparisons. They are combined to form the representative boolean expression BE by the "BoolExp" function, and C is replaced with BE. The ordering of LHS and RHS, along with representing the comparison using LT and EQ, helps generate a consistent boolean expression for the comparison. For example, the condition `(x < y) & & (y > = x)` converts to `LT_x_y & & (LT_x_y || EQ_x_y)`, which simplifies to `LT_x_y` and converts back to `x < y`. The Quine-McCluskey algorithm [24, 31] is used to simplify the boolean expression ET to S. This algorithm does not scale to expressions with a large number of boolean variables. However, based on our assumption that the simplified equations involve only a few comparisons, we typically handle a small number of boolean variables, even though the unsimplified expression may be long and complex with many terms. For each boolean variable B in S, the "GetComparison" function forms the equivalent comparison C. B is replaced with C in S to obtain the final simplified expression.

The equations are further reduced using the Sympy symbolic processing engine [27]. Sympy applies rule-based modifications to simplify or cancel terms in the equation. The definition of simplification is quite subjective, so equations may have more than one representation which can be deemed as simplified. Sympy uses the number of operations as a heuristic to quantify simplification level. Sympy's rule-based simplification fails to apply directly on the complex Angr generated expressions because of the deeply nested conditionals. Hence, the algebraic simplification algorithm is essential to simplify the equations by cleaning up the implementation-specific operations.

Equations (1a)–(1e) show the equations generated by REMaQE for the outputs of `controller_handler`:

$$t = x_8 - \frac{x_1 x_3}{x_4}\left(x_0 - 2x_6 + x_7\right) - x_1 x_2 x_4\left(x_0 - x_5\right) - x_1\left(x_0 - x_6\right) \tag{1a}$$

$$y_0 = \text{round}\left(|y_2|\right) \tag{1b}$$

$$y_1 = |y_2| \tag{1c}$$

$$y_2 = y_5 = \begin{cases} t & \text{for } x_5 - x_0 \le k_2 \text{ and } k_4 \ge t \text{ and } k_3 \le t \\ k_3 & \text{for } x_5 - x_0 \le k_2 \text{ and } k_3 > t \\ k_4 & \text{for } x_5 - x_0 \le k_2 \text{ and } k_3 \le t \text{ and } k_4 < t \\ 0 & \text{otherwise} \end{cases} \tag{1d}$$

$$y_3 = y_4 = \begin{cases} x_0 & \text{for } x_5 - x_0 \le k_2 \\ 0 & \text{otherwise,} \end{cases} \tag{1e}$$

where x_i for $i = 0, \cdots, 8$ are the inputs, y_i for $i = 0, \cdots, 5$ are the outputs, k_i for $i = 2, 3, 4$ are constants, and t is a term introduced to simplify the presentation of equations. The following changes are performed manually to the equations generated by REMaQE: the term t is introduced and replaces the long shared expression in the outputs; the expression for y_2 is replaced in the y_0 and y_1 equations; equations for the repeated outputs y_2, y_5, and y_3, y_4 are displayed together.

4.6 Limitations

REMaQE does not handle data type casting or precision conversion that are intended to perform operations in the original equations (e.g., floor(x) implemented using `float` to `int` cast). Instead, we assume that data type and precision conversions are a by-product of the implementation, and

choose to not represent them in the generated equations to avoid clutter. Similarly, comparison operations that rely on bit manipulation (e.g., $x < 0$ implemented using sign-bit check) are not simplified. These operations are still supported when performed using the proper instructions or library calls. Advanced forms of control-flow like function pointers, recursion, or obfuscated control-flow will run into the path explosion problem of symbolic execution, which will impact the ability to analyze such functions. REMaQE generates equations represented with simple operations on scalar values. So, advanced math operations such as vector dot-product or matrix multiplication are unrolled and represented as a long sequence of individual multiply and add operations. This is less than ideal when the goal is to recover human-friendly equations. Algebraic simplification cannot handle such subjective modifications to the equations to further simplify them, as also seen in Section 4.5 (variable t was separated manually). Appendix A describes additional binaries generated to represent these limitations.

Despite these limitations, REMaQE is applicable to many real-world reverse engineering scenarios. Supporting data type conversions, bit-manipulation, vector and matrix operations, will expand the scope of REMaQE to many more applications, and these are targeted for future work.

5 Evaluation of REMaQE

We focus our evaluation on the widely-used 32-bit ARM architecture with hardware floating point support (ARM32-HF). It is straightforward to extend REMaQE to other architectures supported by symbolic execution frameworks. To evaluate REMaQE, we need to test it on binaries where the implemented math equation is available to check for correctness of the recovered equation. To obtain this dataset for testing, we randomly generate math equations and compile them to ARM32-HF binaries. REMaQE is benchmarked based on correctness defined in Section 5.1, human-friendliness of reverse-engineered equations defined in Section 5.2, and execution time for reverse engineering. Dataset generation is described in Section 5.3.

5.1 Correctness

This section discusses the guarantees of correctness that the framework provides, along with potential sources of inaccuracy. A math equation recovered by REMaQE is "correct" if it is mathematically equivalent to the original equation. Equivalence checking is hard especially when dealing with floating point numbers. Two equations f and g are mathematically equivalent if $f(x) = g(x)\ \forall x \in \mathbb{R}^d$, where d is the number of inputs. As floating point operations are approximate, this strict equality check must be loosened: $\left|\frac{f(x)-g(x)}{f(x)}\right| < \epsilon\ \forall x \in \mathbb{R}^d$, where ϵ is a tolerance to compare the equation outputs. If, $f(x) = 0$, we check if $|g(x)| < \epsilon$. The following sources of inaccuracy arise when implementing equations with floating point variables:

(1) The constants in the original equation are stored in a fixed precision in the binary, hence they can only be recovered to that approximation.
(2) The constants can be modified when the recovered equation is simplified, hence they may not match the original equation constants.
(3) The order of operations can differ between the original equation and recovered equation which impacts floating point computations.

Apart from these inaccuracies, the REMaQE pipeline is deterministic. Angr's symbolic execution, the algebraic simplification algorithm, and Sympy's rule-based modifications preserve and represent the semantic structure of operations present in the binary. This ensures REMaQE always recovers a "correct" equation, even if it is obscured. For the generated dataset where original equations are

available, equivalence check of recovered equations with the original equations is performed on three levels:

(1) *Structural match* is when they have the same order of operations and variables; This is checked by comparing Sympy's internal tree representation of both equations.
(2) *Semantic match* is when they are equivalent, but do not have the same structure; This is checked by taking the difference of equations and seeing if it simplifies to 0.
(3) *Evaluated match* is when the functions are evaluated for a range of inputs and the outputs are compared with tolerance $\epsilon = 10^{-5}$.

A structural match implies semantic match, and a semantic match implies evaluated match. Hence, the equivalence check can stop at the first level that matches. For real world binaries where the original equation is not available, the evaluated match is performed directly. For evaluated match, inputs are randomly selected between -10 and 10. Invalid inputs are discarded if the ground truth equation gives invalid, complex, or infinite outputs. Up to 200 inputs are evaluated to check if the maximum error breaches the threshold. More inputs or a wider range can be chosen for a higher confidence of the evaluated match.

5.2 Human Friendliness

A recovered equation may be considered human-friendly if its complexity is similar to the original equation, based on its appearance. Evaluating the complexity of a math equation is subjective and cannot be precisely measured. So determining whether an equation is fully simplified is not feasible. A heuristic measure of complexity is defined by the number of operations in the math equation. The `sympy.count_ops` function measures the number of operations in an equation. If the reverse engineered equation has a similar number of operations as the original one, we consider them to have comparable complexity. We quantify human-friendliness as the ratio of the number of operations, with a ratio closer to 1 indicating higher human-friendliness.

5.3 Dataset Generation

Each generated equation is implemented as a C function and a Simulink [23] model. Simulink is popular for modelling controllers, and it provides features to compile the models into binaries for embedded targets. The C function is compiled for ARM32-HF target using the GCC compiler (`arm-linux-gnueabihf-gcc`). The Simulink model is compiled for the same target using Simulink's code generation feature. Four optimization levels from "-O0" to "-O3" are used during compilation to obtain a variety of implementations.

To allow direct conversion to Simulink models, which are computational graphs, the math equations are generated as **directed-acyclic graphs (DAG)**, where each node is a parameter or a math operation. The input nodes are initialized first. Then, math operation nodes are picked from a pool of operations, with repeats allowed. Edges are randomly added between the nodes to connect the entire graph, while ensuring that the graph remains a DAG (i.e., no cycles). Finally, nodes without outgoing edges are connected by inserting add or multiply operation nodes to get to one output. This final step prevents "dead" operations (wasted computation not reaching any output).

REMaQE contrasts with approaches in [19, 39] to randomly generate math equations which generate ET instead of DAGs. Generating ET helps control the complexity of the random equation. This is important for the two articles as their analysis uses machine learning and can handle only a finite amount of tokens. REMaQE simplification stage is rule-based and does not have this limitation. In our assessment, DAGs offer two advantages: (1) they can be converted into Simulink models and (2) they generate complex equations in terms of number of operations.

Table 3. Parameters for Random DAG Generation and Binary Compilation

Parameter	DAG Generation Options
Type of operations	Arithmetic, Trigonometric + Exponential, Conditional
Number of inputs	1 to 2
Number of nodes	5 to 15
	Compilation Options
Implementation	C, Simulink
Optimization level	-O0, -O1, -O2, -O3

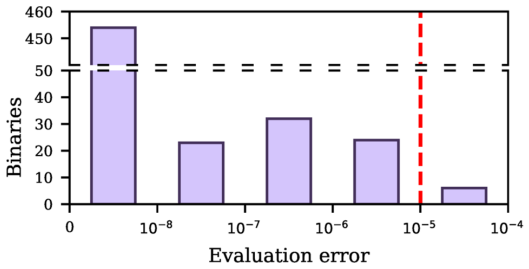

Fig. 4. Histogram of number of binaries vs. max error during *evaluation match*. Tolerance $\epsilon = 10^{-5}$ is marked.

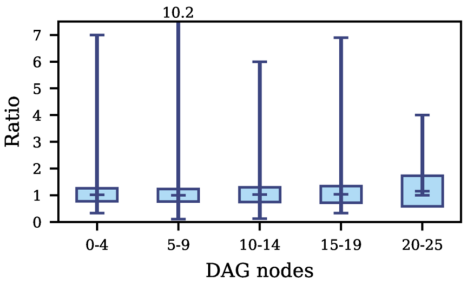

Fig. 5. Average ratio of equation complexity for each parameter binned with the number of nodes. The box indicates $\mu \pm \sigma$ (μ is mean, σ is standard deviation), while the line and whiskers indicate range.

Table 3 shows the parameters available for random DAG generation and compilation. All combinations of parameters are used so that the dataset has a wide variety of equations for evaluation. The nodes inserted to prevent dead operations increase the total number of nodes, so selecting number of nodes up to 15 during generation is sufficient to create DAGs with up to 25 nodes and equations with 100s of operations. The trigonometric and exponential operations generate binaries with calls to library functions, which represents real-world implementations. Conditional operations like saturation, signum, absolute value, and deadzone produce binaries with data-dependent branches and conditional execution that stresses different stages of the REMaQE pipeline.

Each generated equation is simplified first and discarded if the simplification either reduces to a constant, contains complex or infinite values, or cancels out one of the inputs. Each model in the dataset has a generated equation, a simplified ground truth equation, and binaries compiled using C and Simulink implementations with 4 optimization levels. The final dataset contains 3,137 ground truth equations and 25,096 compiled binaries for analysis.

6 Results

REMaQE successfully recovers the correct equation for all 25,096 binaries. This strongly shows the correctness of REMaQE's reverse engineering pipeline. In the equivalence check, *structural match* is obtained for 12,032 binaries (47.94%), *semantic match* is obtained for 12,525 binaries (49.91%), and *evaluated match* is obtained for 533 binaries (2.12%). A tolerance of $\epsilon = 10^{-5}$ is used for the *evaluated match*. The 6 remaining binaries violate the tolerance threshold during evaluation, hence the equivalence check fails for them. However, manual verification (see Section 6.1) reveals that REMaQE has recovered the correct equation for these 6 binaries as well. Figure 4 is a histogram of binaries versus maximum error during *evaluated match*, for the 539 binaries (533 matched plus

Table 4. Examples to Demonstrate the Different Types of Equivalence Checks

Match Type	Ground Truth	Equation Recovered by REMaQE
Structural	$(-k_1 + x_0 \sin(k_0 x_1)^2 - x_1)/k_2$	$(-k_1 + x_0 \sin(k_0 x_1)^2 - x_1)/k_2$
Semantic	$(-k_0 + k_1 + k_2 x_0 - x_1 - \text{atan}(x_0))\exp(-x_0)$	$\frac{-\text{atan}(x_0)+x_0 k_2+k_1-x_1-k_0}{\exp(x_0)}$
Evaluated	$\text{acos}(k_1 x_0) - \frac{x_1 \sin(x_0)}{k_0}$	$\begin{cases} \text{acos}(k_1 x_0) - x_1 \sin(x_0)/k_0 \\ \quad \text{for } (k_1 x_0 >= -1.0) \text{ and } (k_1 x_0 <= 1.0) \\ 3.141593 - x_1 \sin(x_1)/k_0 \\ \quad \text{for } k_1 x_1 <= 1.0 \\ -x_1 \sin(x_0)/k_0 \\ \quad \text{otherwise} \end{cases}$

Even in instances such as the one shown in the "Evaluated" row, REMaQE recovers the correct representation of equations implemented in the binary (which may differ from "Ground Truth") since the compilation tool chain adds these checks.

6 manually verified). Most binaries have a maximum error $\leq 10^{-8}$, indicating the equations are correct with confidence. Very few have higher errors and even fewer breach the tolerance threshold due to floating point approximations.

Table 4 shows examples from the dataset for each of the three equivalence check levels. The recovered equation for the *evaluated match* case is much more complex than the ground truth, with extra conditional operations, because Simulink adds bounds checks for the acos(·) function. REMaQE reverse engineers the implementation in the binary, hence semantic modifications performed during compilation, such as the extra bounds checks introduced by Simulink in this case, are also recovered. The recovered equations help debug the implementation of mathematical algorithms to ensure that the compilation pipeline has not made any unexpected changes. Evaluating both equations shows that they generate the same outputs (or close, up to the tolerance).

6.1 Manual Verification

The goal of REMaQE is to reverse engineer the binary implementation and present it as math equations. When the three equivalence checks fail to match the recovered and ground truth equations, this is not sufficient to conclude that the recovered equation is wrong. Manual inspection is required to determine whether the recovered equation is a faithful recreation of the implementation of the ground truth equation, or if it is indeed mismatched. The following is one of the 6 manually verified cases to demonstrate that even though equivalence checks have failed, REMaQE has recovered correct equations. Equation (2a) shows the ground truth and Equation (2b) shows the recovered equation:

$$y = \begin{cases} x - k_0 - k_1 - \frac{1}{k_0^2} - \frac{x}{k_0} + \frac{1}{k_0} & \text{for } x > 0 \\ x - k_0 - k_1 - \frac{1}{k_0^2} + \frac{x}{k_0} + \frac{1}{k_0} & \text{for } x < 0 \\ x - k_0 - k_1 & \text{otherwise} \end{cases} \tag{2a}$$

$$y = \begin{cases} -\tilde{k_0} - \tilde{k_1} & \text{for } x = 0 \\ x - \tilde{k_0} - \tilde{k_1} - \tilde{k_2}\left(x - \tilde{k_0} + \tilde{k_2} + 1\right) + 1 & \text{for } x < 0 \\ x - \tilde{k_0} - \tilde{k_1} - 1 + \frac{\tilde{k_0} - x + 1 - \frac{1}{\tilde{k_0}}}{\tilde{k_0}} & \text{for } x > 0 \end{cases} \tag{2b}$$

where x is the input, y is the output, k_i are the constants in the ground truth equation, and $\tilde{k_i}$ are the approximated constants recovered from the binary. $k_0 = -1.51$, $k_1 = -1.58$, $\tilde{k_0} =$

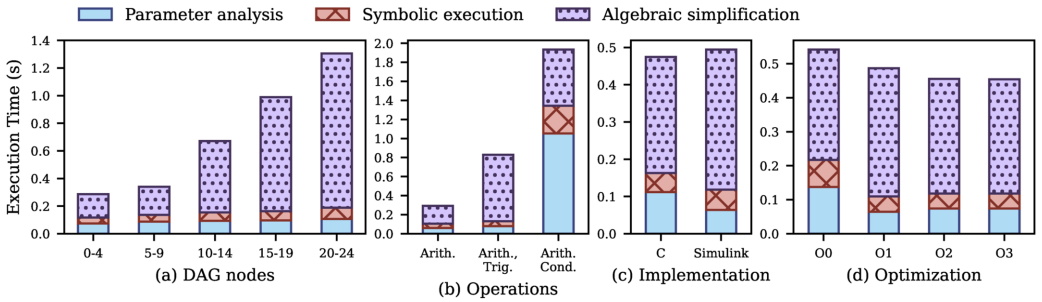

Fig. 6. Average execution time (seconds) per stage for the different dataset generation and compilation parameters: (a) number of DAG nodes, (b) type of operations, (c) implementation, and (d) optimization level.

-1.5099999904632568, $\tilde{k_1} = -1.5800000429153442$, and $\tilde{k_2} = -0.6622516512870789$. Notice that Equation (2b) has an extra constant $\tilde{k_2} \approx \frac{1}{k_0}$, which means the compiler has chosen to store the inverse of k_0 as a separate constant. Inaccuracies due to the floating point approximation and different representation of constants caused *evaluated match* to find inputs that violate the tolerance level. Manual inspection shows that REMaQE has recovered a semantically matched equation.

6.2 Human Friendliness

Figure 5 demonstrates the human-friendliness of reverse engineered outputs of REMaQE. The average ratio of number of operations in the recovered equation over the simplified ground truth equation is shown as a histogram binned with respect to number of nodes in the DAG. Standard deviation and range of each bin is displayed as the box and whiskers plot. The average ratio is close to 1 across the range of equation complexity, with a standard deviation of 0.26; For larger number of nodes, the mean increases to 1.15 and standard deviation increases to 0.57. REMaQE performs well on the human-friendliness metric for a variety of equations. Instances with higher ratios indicate outliers where REMaQE recovers a more complex equation since algebraic simplification cannot shrink the recovered equations beyond a point. Instances with ratios lower than 1 indicate cases where the recovered equations are simpler than ground truth since C and Simulink compilers optimized the expressions in the implementation.

6.3 Execution Time

Execution time of each of REMaQE's three stages is measured for the generated dataset on an Intel i7-6700 CPU. Only one instance of REMaQE is run on a single core to collect the timing measurements. On average, the full reverse engineering takes 0.48 seconds. 71% time is consumed by algebraic simplification, 18% by parameter analysis, and 11% by symbolic execution.

Figure 6 displays the execution time for different dataset generation parameters. Across the different kinds of binaries, the total execution time remains under 2 seconds.

(a) *DAG nodes*: Execution time for all three stages increases almost linearly with number of nodes in the generated DAG. This is because code size (i.e., number of instructions) is proportional to DAG nodes. Execution time ranges from 0.3 seconds to 1.3 seconds across DAG nodes.

(b) *Type of operations*: We see a significant increase for the more complex operations. Trigonometric and exponential operations cause a larger number of function calls to libraries, taking longer to reverse engineer. Conditional operations add data-dependent branches that lead to multiple exploration paths during parameter analysis and symbolic execution. Each branch execution adds conditions to the final equation that require additional handling during algebraic simplification.

```
void PosVelEKF::predict(float dt, float dVel, float dVelNoise)
{
    // Newly predicted state and covariance matrix at next time step
    float newState[2];
    float newCov[3];
    // We assume the following state model for this problem
    newState[0] = dt*_state[1] + _state[0];
    newState[1] = dVel + _state[1];
    /* ... */
    newCov[0] = dt*_cov[1] + dt*(dt*_cov[2] + _cov[1]) + _cov[0];
    newCov[1] = dt*_cov[2] + _cov[1];
    newCov[2] = ((dVelNoise)*(dVelNoise)) + _cov[2];
    // store the predicted matrices
    memcpy(_state,newState,sizeof(_state));
    memcpy(_cov,newCov,sizeof(_cov));
}

float calc_lowpass_alpha_dt(float dt, float cutoff_freq)
{
    if (is_negative(dt) || is_negative(cutoff_freq)) {
        INTERNAL_ERROR(AP_InternalError::error_t::invalid_arg_or_result);
        return 1.0;
    }
    if (is_zero(cutoff_freq)) return 1.0;
    if (is_zero(dt)) return 0.0;
    float rc = 1.0f / (M_2PI * cutoff_freq);
    return dt / (dt + rc);
}
```

Fig. 7. C++ source of `PosVelEKF::predict` and `calc_lowpass_alpha_dt`.

(c) *Implementation*: Parameter analysis and symbolic execution are slightly faster for Simulink compared to C, whereas algebraic simplification is slightly slower. This is because Simulink may add extra bounds checks for some operations that take longer to process during algebraic simplification.

(d) *Optimization level*: For higher optimization levels, parameter analysis and symbolic execution time decreases, as the code size decreases.

Overall, REMaQE reverse engineers a wide range of equations in real time. It is hence suitable as an interactive tool that can integrate into a GUI-based reverse engineering workflow and deliver recovered equations at latencies comparable to human click speeds.

7 Case Study: Reverse Engineering of ArduPilot

We use REMaQE to reverse engineer two functions in the ArduPilot auto-pilot firmware for UAVs [3]. The firmware binary[4] is compiled for ARM32-HF and the BeagleBone Black embedded platform. The binary is not stripped in this case, so function names and global symbol names are available. Figure 7 shows the C++ source for `PosVelEKF::predict` which implements an extended Kalman filter (EKF) for position-velocity estimation and `calc_lowpass_alpha_dt`, a low-pass filter. We reverse engineer these functions using REMaQE. The source code is provided only for discussion and is not utilized during reverse engineering.

Table 5 shows the parameter metadata extracted for `PosVelEKF::predict`. REMaQE correctly identifies the 3 pass-by-value arguments (x_0 to x_2) and 5 class members (x_5 to x_9) as inputs of the function. The missing indices are false positive inputs identified by REMaQE. They do not show up in the output equations, and they are not displayed in the table. The class members are also identified as the 5 outputs (y_0 to y_4). The class pointer is correctly identified as the base for the class members and represented as `ptr0`. REMaQE has handled the `memcpy` call and determined outputs even though the function returns `void`.

Equations (3a)–(3e) show the recovered output equations:

$$y_0 = x_0 x_6 + x_5, \tag{3a}$$

[4] https://firmware.ardupilot.org/Copter/stable-4.3.4/bbbmini/arducopter

Table 5. Parameters of `PosVelEKF::predict`

Inp.	Kind	Location	Size
x_0	register	`s0`	32
x_1	register	`s1`	32
x_2	register	`s2`	32
x_5	pointer	`ptr0[0x0]`	32
x_6	pointer	`ptr0[0x4]`	32
x_7	pointer	`ptr0[0x8]`	32
x_8	pointer	`ptr0[0xc]`	32
x_9	pointer	`ptr0[0x10]`	32

Out.	Kind	Location	Size
y_0	pointer	`ptr0[0x0]`	32
y_1	pointer	`ptr0[0x4]`	32
y_2	pointer	`ptr0[0x8]`	32
y_3	pointer	`ptr0[0xc]`	32
y_4	pointer	`ptr0[0x10]`	32
Ptr.	**Kind**	**Location**	**Size**
`ptr0`	register	`r0`	32

Table 6. Parameters of `calc_lowpass_alpha_dt`

Inp.	Kind	Location	Size
x_0	register	`s0`	32
x_1	register	`s1`	32
Out.	**Kind**	**Location**	**Size**
y_0	register	`s0`	32

Const.	Kind	Location	Size	Value
k_0	global	`0x473f88`	64	6.28319
k_1	global	`0x473f90`	32	−1.19209e-07
k_2	global	`0x473f94`	32	1.19209e-07

$$y_1 = x_1 + x_6, \tag{3b}$$

$$y_2 = x_0 x_8 + x_0 \left(x_0 x_9 + x_8\right) + x_7, \tag{3c}$$

$$y_3 = x_0 x_9 + x_8, \tag{3d}$$

$$y_4 = x_2^2 + x_9, \tag{3e}$$

where x_i for $i = 0, 1, 2, 5, \cdots, 9$ are inputs and y_i for $i = 0, \cdots, 4$ are outputs. REMaQE reverse engineers the math equations implemented in the function. While these equations by themselves are not sufficient to determine that this function implements an EKF, the user can combine them with context from other sources to understand aspects like the state and covariance updates.

Table 6 shows the parameter metadata extracted for `calc_lowpass_alpha_dt`. The error function is ignored during symbolic execution so REMaQE captures the error cases without terminating. REMaQE identifies the pass-by-value arguments and return output. The recovered constant values are helpful: $k_0 \approx 2\pi$, also $k_1 \approx -2^{-23}$ and $k_2 \approx 2^{-23}$, the 32-bit floating point machine epsilons.

Equation (4) shows the output equation:

$$y_0 = \begin{cases} 1 & \text{for } k_1 \le x_0 \text{ and } k_1 \le x_1 \text{ and } k_2 > |x_1| \\ 0 & \text{for } k_2 \ge |x_0| \text{ and } k_1 \le x_0 \text{ and } k_1 \le x_1 \text{ and } k_2 \le |x_1| \\ \frac{k_0 x_0 x_1}{k_0 x_0 x_1 + 1} & \text{for } k_1 \le x_0 \text{ and } k_1 \le x_1 \text{ and } k_2 \le |x_1| \text{ and } k_2 < |x_0| \\ 1 & \text{otherwise} \end{cases}, \tag{4}$$

where y_0 is the output, x_0, x_1 are the inputs, and k_0, k_1, k_2 are constants. The expression for low-pass filter is shown in one of the cases. The conditions in case statements indicate that `is_zero()` and `is_negative()` in the original code are implemented as comparisons with k_1 and k_2, the machine epsilon, instead of relying on equality or sign-bit check. Using REMaQE, we verify that the implementation is robust to floating point approximations. This case study demonstrates that REMaQE can be used in real-world analysis of UAVs, and debugging of implemented algorithms.

```
FUNCTION_BLOCK MYDERIVATIVE
  VAR_INPUT xin:REAL; cycle:REAL; END_VAR
  VAR_OUTPUT xout:REAL; END_VAR
  VAR x1, x2, x3:REAL; END_VAR

  xout := (3.0 * (xin - x3) + x1 - x2)
          / (10.0 * cycle);
  x3 := x2; x2 := x1; x1 := xin;
END_FUNCTION_BLOCK

FUNCTION_BLOCK MYINTEGRAL
  VAR_INPUT xin:REAL; cycle:REAL; END_VAR
  VAR_OUTPUT xout:REAL; END_VAR

  xout := xout + xin * cycle;
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK MYPID
  VAR_INPUT pv:REAL; setp:REAL; cycle:REAL; END_VAR
  VAR_OUTPUT xout:REAL; END_VAR
  VAR
    Kp:REAL; Tr:REAL; Td:REAL; error:REAL;
    iterm:MYINTERGRAL; dterm:MYDERIVATIVE;
  END_VAR

  Kp := 1.54; Tr := 2.33; Td := 0.07;
  error := pv - setp;
  iterm (xin := error, cycle := cycle);
  dterm (xin := error, cycle := cycle);
  xout := Kp * (error + iterm.xout/Tr
               + dterm.xout*Td);
END_FUNCTION_BLOCK
```

Fig. 8. Structured Text implementation of a PID controller for OpenPLC.

Table 7. Parameters of the OpenPLC PID Controller

Inp.	Kind	Location	Size
x_0	pointer	`ptr0[0x0]`	32
x_1	pointer	`ptr0[0x8]`	32
x_2	pointer	`ptr0[0x10]`	32
x_3	pointer	`ptr0[0x18]`	32
x_4	pointer	`ptr0[0x60]`	32
x_5	pointer	`ptr0[0x80]`	32
x_6	pointer	`ptr0[0x88]`	32
x_7	pointer	`ptr0[0x90]`	32
Ptr.	**Kind**	**Location**	**Size**
`ptr0`	register	`r0`	32

Out.	Kind	Location	Size	
y_0	pointer	`ptr0[0x20]`	32	
y_4	pointer	`ptr0[0x60]`	32	
y_5	pointer	`ptr0[0x80]`	32	
y_6	pointer	`ptr0[0x88]`	32	
y_7	pointer	`ptr0[0x90]`	32	
Const.	**Kind**	**Location**	**Size**	**Value**
k_0	immediate	`0x40019b`	32	1.54
k_1	immediate	`0x4001af`	32	2.33
k_2	immediate	`0x4001c3`	32	0.07

8 Case Study: Reverse Engineering of the OpenPLC PID Controller

We use REMaQE to reverse engineer a PID controller implemented using the OpenPLC [2] platform in the **Structured Text (ST)** language. OpenPLC converts the ST implementation to C code, which is compiled to an object file for the ARM32-HF target using the GCC compiler. This object file can be executed using the OpenPLC runtime on embedded devices that OpenPLC supports, such as the Arduino and Raspberry Pi. Figure 8 shows the ST source code of the PID controller. The function blocks in this ST are based on the "PID", "INTEGRAL", and "DERIVATIVE" examples provided in the IEC 61131-3 standard [9]. The source code is provided here only for discussion and is not used during reverse engineering. REMaQE is run directly on the "MYPID" function as we intend to inline the integral and derivative calculations in the recovered equations instead of representing them as function calls. Table 7 shows the parameters of the PID controller. We see that access to input, output and local variables are pointer based. We also see the three tuning constants used by the controller, whose usage will be evident from the recovered equations. The inputs x_4, x_5, x_6, x_7 and outputs y_4, y_5, y_6, y_7 are correspondingly accessed from the same location (same pointer offset), indicating that they may refer to the local variables.

Equations (5a)–(5e) show the recovered output equations:

$$y_0 = k_0 \left(x_0 - x_1 + \frac{1}{k_1} \left(x_4 + x_3 \left(x_0 - x_1 \right) \right) + \frac{0.1 k_2}{x_3} \left(x_5 - x_6 + 3.0 \left(x_0 - x_1 - x_7 \right) \right) \right) \tag{5a}$$

$$y_4 = x_4 + x_3 \left(x_0 - x_1 \right) \tag{5b}$$

$$y_5 = x_0 - x_1 \tag{5c}$$

$$y_6 = x_5 \tag{5d}$$

$$y_7 = x_6 \tag{5e}$$

where x_i for $i = 0, \cdots, 7$ are inputs, y_i for $i = 0, 4, \cdots, 7$ are outputs, and k_0, k_1, k_2 are constants. The equation for y_0 looks like the PID output, which means that k_0 must be "Kp". The expression $x_0 - x_1$ is repeated in multiple places, indicating that it is the error term, making x_0 as process value "pv" and x_1 as setpoint "setp". As x_4 and y_4 are stored in the same location, Equation (5b) represents an increment of x_4 with $x_3(x_0 - x_1)$, so it is likely the integral accumulator. This means the second term in Equation (5a) is the integral term and the third is the derivative term. So, y_5, y_6, y_7 store previous error values in shifted manner, to be used in the four-point derivative approximation term. We can also conclude that k_1 is "Tr", k_2 is "Td", and x_3 is the cycle time input. These findings match the ST implementation in Figure 8. Notice that x_2 is identified as an input, but is not used in the equations. This happens when some locations are accessed for other purposes, but parameter analysis marks them as inputs of the equations. This case study demonstrates that REMaQE can be used to reverse engineer PLC binary executables, which helps to recover source code of legacy systems, and verify the integrity of implemented equations to ensure there is no tampering.

9 Conclusion

The REMaQE framework automatically reverse engineers math equations from binary executables. It has three stages: parameter analysis, symbolic execution, and algebraic simplification. Parameter analysis allows REMaQE to identify the input, output and constant parameters of the math equation and whether they are stored in the register, on the stack, in global memory or accessed via pointers. REMaQE uses this metadata to reverse engineer a wide variety of implementations, such as C++ classes that use class pointers, or Simulink compiled binaries that use global memory. This is a significant improvement over the state-of-the-art, which only handles simple functions implemented using registers. Algebraic simplification allows REMaQE to simplify complex conditional expressions involving floating-point comparisons, that are generated by symbolic execution even for simple equations. Existing approaches use machine learning models that have limits on the input expression size. They cannot handle the long, complex expressions generated by conditional computations.

Additionally, we introduce an alternative method for randomly generating diverse equations using directed acyclic graphs. REMaQE recovers the correct equations for the entire dataset of 25,096 binaries. REMaQE takes an average execution time of 0.48 seconds and up to 2 seconds for the complex equations. Such a small reverse engineering time makes REMaQE effective in an interactive reverse engineering workflow, and would enable one-click latency when integrated with GUI frameworks. Comparing the complexity of the recovered equations with the original equations, REMaQE shows an average ratio of 1 through a range of equation complexities, with a standard deviation of 0.26. The REMaQE automated tool can significantly reduce human effort in the recovery of control system dynamics from legacy hardware or recovered adversaries' control computers and can also facilitate defenses during run-time or after an attack to determine if any parameters or control dynamics are modified. This can also reveal the effects of the modifications on the mathematical computations and therefore system performance. REMaQE extracts human-friendly equations.

Future work can extend REMaQE to integrate it into GUI decompilation tools, handle data type conversion and bitwise computation, bundle support for complex control-flow like function pointers or recursion, and enhance reverse engineering capabilities to recognize advanced mathematical structures for improved representation. Algorithms in embedded systems can employ advanced mathematical structures such as dynamic loops, summation, integration, differentiation, and high dimensional data such as vectors and matrices. Extending simplification to include long chains of

scalar computation into high-level operations will further improve REMaQE's human-friendliness. These refinements and extensions are the proposed future work for REMaQE.

References

[1] Riham Altawy and Amr M. Youssef. 2016. Security, privacy, and safety aspects of civilian drones: A survey. *ACM Transactions on Cyber-Physical Systems* 1, 2 (Nov. 2016), Article 7, 25 pages.

[2] Thiago Alves. 2023. OpenPLC. Retrieved December 20, 2023 from https://autonomylogic.com/

[3] ArduPilot. 2023. ArduPilot. Retrieved December 20, 2023 from https://ardupilot.org/

[4] C. W. Badenhop, S. R. Graham, B. E. Mullins, and L. O. Mailloux. 2018. Looking under the hood of Z-Wave: Volatile memory introspection for the ZW0301 Transceiver. *ACM Transactions on Cyber-Physical Systems* 3, 2 (Dec. 2018), Article 20, 24 pages.

[5] Hamza Bourbouh, Pierre-Loïc Garoche, Christophe Garion, and Xavier Thirioux. 2021. *From Lustre to Simulink: Reverse Compilation for Embedded Systems Applications. ACM Transactions on Cyber-Physical Systems* 5, 3 (Jul. 2021), Article 31, 20 pages.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 209–224.

[7] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, USA, 380–394.

[8] Cristina Cifuentes. 1994. *Reverse Compilation Techniques*. Ph.D. Dissertation. Queensland University of Technology.

[9] International Electrotechnical Commission. 2003. IEC 61131-3:2003. Retrieved December 20, 2023 from https://webstore.iec.ch/publication/19081

[10] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, Japan, 653–656.

[11] Hex-Rays. 2023. IDA Pro Disassembler. Retrieved December 20, 2023 from https://hex-rays.com/ida-pro/

[12] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, 215–224.

[13] Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed. 2019. CLIK on PLCs! Attacking control logic with decompilation and virtual PLC. In *Proceedings of the Workshop on Binary Analysis Research (BAR)*. The Internet Society, 74–85.

[14] Anastasis Keliris and Michail Maniatakos. 2019. ICSREF: A framework for automated reverse engineering of industrial control systems binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 271–285.

[15] Farshad Khorrami, Prashanth Krishnamurthy, and Ramesh Karri. 2016. Cybersecurity for control systems: A process-aware perspective. *IEEE Design & Test* 33, 5 (2016), 75–83.

[16] Taegyu Kim, Aolin Ding, Sriharsha Etigowni, Pengfei Sun, Jizhou Chen, Luis Garcia, Saman Zonouz, Dongyan Xu, and Dave (Jing) Tian. 2022. Reverse engineering and retrofitting robotic aerial vehicle control firmware using dispatch. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22)*. ACM, New York, NY, 69–83.

[17] Charalambos Konstantinou, Michail Maniatakos, Fareena Saqib, Shiyan Hu, Jim Plusquellic, and Yier Jin. 2015. Cyber-physical systems: A security perspective. In *Proceedings of the IEEE European Test Symposium (ETS)*. IEEE, USA, 1–8.

[18] Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, David Paul-Pena, and Hossein Salehghaffari. 2018. Process-aware covert channels using physical instrumentation in cyber-physical systems. *IEEE Transactions on Information Forensics and Security* 13, 11 (2018), 2761–2771.

[19] Guillaume Lample and François Charton. 2020. Deep learning for symbolic mathematics. In *Proceedings of the International Conference on Learning Representations (ICLR)*. OpenReview.net, 24 pages.

[20] Ralph Langner. 2011. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy* 9, 3 (2011), 49–51.

[21] Robert M. Lee, Michael J. Assante, and Tim Conway. 2016. Analysis of the cyber attack on the Ukrainian power grid. Retrieved from https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2016/05/20081514/E-ISAC_SANS_Ukraine_DUC_5.pdf.

[22] Xuefeng Lv, Yaobin Xie, Xiaodong Zhu, and Lu Ren. 2017. A technique for bytecode decompilation of PLC program. In *Proceedings of the Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. IEEE, 252–257.

[23] MathWorks. 2023. Simulink - Simulation and Model-Based Design. Retrieved December 20, 2023 from https://www.mathworks.com/products/simulink.html
[24] E. J. McCluskey Jr. 1956. Minimization of Boolean functions. *The Bell System Technical Journal* 35, 6 (1956), 1417–1444.
[25] Stephen McLaughlin. 2011. On dynamic malware payloads aimed at programmable logic controllers. In *Proceedings of the USENIX Conference on Hot Topics in Security*. USENIX Association, 10–15.
[26] Stephen McLaughlin, Saman Zonouz, Devin Pohly, and Patrick McDaniel. 2014. A trusted safety verifier for process controller code. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 43–57.
[27] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, et al. 2017. SymPy: Symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103.
[28] NSA. 2023. Ghidra. Retrieved December 20, 2023 from https://ghidra-sre.org/
[29] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. 2011. Obfuscation: The hidden malware. *IEEE Security & Privacy* 9, 5 (2011), 41–47.
[30] Syed Ali Qasim, Juan Lopez, and Irfan Ahmed. 2019. Automated reconstruction of control logic for programmable logic controller forensics. In *Information Security*. Zhiqiang Lin, Charalampos Papamanthou, and Michalis Polychronakis (Eds.), Springer International Publishing, Cham, 402–422.
[31] W. V. Quine. 1952. The problem of simplifying truth functions. *The American Mathematical Monthly* 59 (1952), 521–531.
[32] Saranyan Senthivel, Shrey Dhungana, Hyunguk Yoo, Irfan Ahmed, and Vassil Roussev. 2018. Denial of engineering operations attacks in industrial control systems. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY '18)*. ACM, New York, NY, 319–329.
[33] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice – automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 294–308.
[34] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. 2016. SOK: (State of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.
[35] Nick Stephens, Jessie Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 368–383.
[36] Pengfei Sun, Luis Garcia, and Saman Zonouz. 2019. Tell me more than just assembly! Reversing cyber-physical execution semantics of embedded IoT controller software binaries. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 349–361.
[37] R. Sun, A. Mera, L. Lu, and D. Choffnes. 2021. SoK: Attacks on industrial control logic and formal verification-based defenses. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS & P)*. IEEE Computer Society, 385–402.
[38] Meet Udeshi, Prashanth Krishnamurthy, Hammond Pearce, Ramesh Karri, and Farshad Khorrami. 2023. Dataset for REMaQE evaluation. IEEE Dataport. DOI: https://dx.doi.org/10.21227/r7e6-bk48
[39] Nicolaas Weideman, Virginia K. Felkner, Wei-Cheng Wu, Jonathan May, Christophe Hauser, and Luis Garcia. 2021. PERFUME: Programmatic extraction and refinement for usability of mathematical expression. In *Proceedings of the Research on Offensive and Defensive Techniques in the Context of Man at the End (MATE) Attacks (Checkmate)*. ACM, New York, NY, 59–69.
[40] Zeyu Yang, Liang He, Hua Yu, Chengcheng Zhao, Peng Cheng, and Jiming Chen. 2022. Detecting PLC intrusions using control invariants. *IEEE Internet of Things Journal* 9, 12 (2022), 9934–9947.

Appendices

A Additional Dataset Representing Limitations

REMaQE is able to successfully handle all the cases in the dataset of 3,137 equations and 25,096 binaries. However, to illustrate cases representing the limitations of REMaQE, we have generated 212 additional binaries in a separate folder in the dataset. In these binaries, we have made the following two implementation changes:

—A new signum implementation using bitwise operations to check the floating point sign bit
—Both 32-bit and 64-bit floating point types are used in each function to introduce type-casting

Only the C implementation is used for compilation as these changes cannot be introduced into Simulink's compilation pipeline. On these binaries, REMaQE recovers the accurate equation for 94

(44.34%), fails for 35 (16.51%), and takes more than 10 minutes to run for 83 (39.15%). The algebraic simplification time blows up exponentially for complex expressions as those generated by bitwise operations, and may sometimes generate wrong simplifications. The type-casting produces both 32-bit and 64-bit load and store operations that confuses the parameter analysis regarding the bit-width of a parameter, which leads to wrong equations.

B Simplification Example

$$y = \begin{cases} k_0 - k_1 - k_2\,(k_2 - x_0) + k_2 + k_4 x_0 - x_0 & \text{for } k_2 > k_4 + x_0 \\ k_0 - k_1 - k_2\,(k_2 - x_0) + k_2 + k_3 x_0 - x_0 & \text{for } k_2 < k_3 + x_0 \\ k_0 - k_1 - k_2\,(k_2 - x_0) + k_2 + x_0\,(k_2 - x_0) - x_0 & \text{otherwise} \end{cases}$$

(a) Ground Truth equation from the dataset.

```
<FP32 if (1 & ~(LShR(LShR((((LShR((if fpLT(fpSub(k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then
0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))), 0x5) & 0x3 | (if fpLT(fpSub(k2, x0), k4) then 0
x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))) & 0x1) ^
 0x1) << 0x1e) - 0x1, 0x1d) + 0x1 - ((LShR((if fpLT(fpSub(k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0),
k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))), 0x5) & 0x3 | (if fpLT(fpSub(k2, x0), k4
) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45)))
& 0x1) & LShR(LShR((if fpLT(fpSub(k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if
fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))), 0x5) & 0x3 | (if fpLT(fpSub(k2, x0), k4) then 0x1 else (if
fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))) & 0x1, 0x1) & 0x1)
<< 0x1c & 0xf0000000, 0x1e)[0:0] & 1 | LShR(LShR((((LShR((if fpLT(fpSub(k2, x0), k4) then 0x1 else (if fpGT(
fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))), 0x5) & 0x3 | (if fpLT(
fpSub(k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0
x40 else 0x45))) & 0x1) ^ 0x1) << 0x1e) - 0x1, 0x1d) + 0x1 - ((LShR((if fpLT(fpSub(k2, x0), k4) then 0x1 else
 (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))), 0x5) & 0x3 | (
if fpLT(fpSub(k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4
) then 0x40 else 0x45))) & 0x1) & LShR(LShR((if fpLT(fpSub(k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0),
 k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))), 0x5) & 0x3 | (if fpLT(fpSub(k2, x0),
k4) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))
) & 0x1, 0x1) & 0x1) << 0x1c & 0xf0000000, 0x1f)[0:0] & 1 ^ LShR(LShR((((LShR((if fpLT(fpSub(k2, x0), k4)
then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))), 0
x5) & 0x3 | (if fpLT(fpSub(k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(
fpSub(k2, x0), k4) then 0x40 else 0x45))) & 0x1) ^ 0x1) << 0x1e) - 0x1, 0x1d) + 0x1 - ((LShR((if fpLT(fpSub(
k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40
else 0x45))), 0x5) & 0x3 | (if fpLT(fpSub(k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0
else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))) & 0x1) & LShR(LShR((if fpLT(fpSub(k2, x0), k4) then 0
x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2, x0), k4) then 0x40 else 0x45))), 0x5) &
0x3 | (if fpLT(fpSub(k2, x0), k4) then 0x1 else (if fpGT(fpSub(k2, x0), k4) then 0x0 else (if fpEQ(fpSub(k2,
x0), k4) then 0x40 else 0x45))) & 0x1, 0x1) & 0x1) << 0x1c & 0xf0000000, 0x1c)[0:0] & 1)) == 0 then fpAdd(
fpAdd(k0, fpNeg(fpAdd(fpAdd(fpNeg(fpAdd(k2, fpNeg(x0))), fpMul(fpAdd(k2, fpNeg(x0)), k2)), k1))), fpMul(x0, (
if 1 + (0 .. 0xffffffff + ((if fpLT(fpAdd(k2, fpNeg(x0)), k3) || fpLT(k3, fpAdd(k2, fpNeg(x0))) then 0 else
1) .. ~(if fpLT(fpAdd(k2, fpNeg(x0)), k3) then 1 else (if fpEQ(fpAdd(k2, fpNeg(x0)), k3) || fpLT(k3, fpAdd(k2
, fpNeg(x0))) then 0 else 1)) .. 0x0)[31:29]) + 15 * ~(~((0 .. (if fpLT(fpAdd(k2, fpNeg(x0)), k3) || fpLT(k3,
 fpAdd(k2, fpNeg(x0))) then 0 else 2)) | (0 .. (if fpLT(fpAdd(k2, fpNeg(x0)), k3) then 1 else (if fpEQ(fpAdd(
k2, fpNeg(x0)), k3) || fpLT(k3, fpAdd(k2, fpNeg(x0))) then 0 else 1)))) | (7 .. ~(if fpLT(fpAdd(k2, fpNeg(x0)
), k3) || fpLT(k3, fpAdd(k2, fpNeg(x0))) then 0 else 1)) | 14)[3:3] == 1 then k3 else fpAdd(k2, fpNeg(x0)))))
 else fpAdd(fpAdd(k0, fpNeg(fpAdd(fpAdd(fpNeg(fpAdd(k2, fpNeg(x0))), fpMul(fpAdd(k2, fpNeg(x0)), k2)), k1))),
 fpMul(x0, k4))>
```

(b) The unsimplified ET. Long symbol names are replaced and redundant information is removed to truncate the ET to fit. The full ET is 17000 characters long.

$$y = \begin{cases} k_0 + k_4 x_0 - (k_1 + k_2\,(k_2 - x_0) - k_2 + x_0) & \\ \quad \text{for } k_4 < k_2 - x_0 & \\ \begin{cases} k_0 + k_3 x_0 - (k_1 + k_2\,(k_2 - x_0) - k_2 + x_0) & \text{for } k_3 > k_2 - x_0 \\ k_0 + x_0\,(k_2 - x_0) - (k_1 + k_2\,(k_2 - x_0) - k_2 + x_0) & \text{otherwise} \end{cases} & \\ \quad \text{otherwise} & \end{cases}$$

(c) Equation generated after Quine-McCluskey simplification.

$$y = \begin{cases} k_0 - k_1 - k_2\,(k_2 - x_0) + k_2 + k_4 x_0 - x_0 & \\ \quad \text{for } k_2 > k_4 + x_0 & \\ k_0 - k_1 - k_2\,(k_2 - x_0) + k_2 + k_3 x_0 - x_0 & \\ \quad \text{for } k_2 < k_3 + x_0 & \\ k_0 - k_1 - k_2\,(k_2 - x_0) + k_2 + x_0\,(k_2 - x_0) - x_0 & \\ \quad \text{otherwise} & \end{cases}$$

(d) Final simplified equation.

Fig. B1. An example to show unsimplified and simplified equations: (a) ground truth equation, (b) the unsimplified ET obtained after parameter analysis and symbolic execution, (c) the equation generated after applying the Quine-McCluskey algebraic simplification on the ET, and (d) the final simplified equation produced after Sympy simplification.

Received 18 January 2024; revised 18 June 2024; accepted 24 September 2024