PIE: Parser Identification in Embedded Systems

Lucian Cojocar[*]
l.cojocar@vu.nl

Jonas Zaddach[†]
jonas.zaddach@eurecom.fr

Roel Verdult[‡]
rverdult@cs.ru.nl

Herbert Bos[*]
herbertb@cs.vu.nl

Aurélien Francillon[†]
aurelien.francillon@eurecom.fr

Davide Balzarotti[†]
davide.balzarotti@eurecom.fr

ABSTRACT

Embedded systems are responsible for the security and safety of modern societies, controlling the correct operation of cars and airplanes, satellites and medical equipment, military units and all critical infrastructures. Being integrated in large and complex environments, embedded systems need to support several communication protocols to interact with other devices or with their users. Interestingly, embedded software often implements protocols that deviate from their original specifications. Some are extended with additional features, while others are completely undocumented. Furthermore, embedded parsers often consist of complex C code which is optimized to improve performance and reduce size. However, this code is rarely designed with security in mind, and often lacks proper input validation, making those devices vulnerable to memory corruption attacks. Furthermore, most embedded designs are closed source and third party security evaluations are only possible by looking at the binary firmware.

In this paper we propose a methodology to identify parsers and complex processing logic present in binary code without access to their source code or documentation. Specifically we establish and evaluate a heuristic for detecting this type of code by means of static analysis. Afterwards we demonstrate the utility of this heuristic to identify firmware components treating input, perform reverse engineering to extract protocols, and discover and analyze bugs on four widely used devices: a GPS receiver, a power meter, a hard disk drive (HDD) and a Programmable Logic Controller (PLC).

1. INTRODUCTION

Embedded devices are more and more present in our everyday lives. While we rely on them for our safety and security, the frequent vulnerabilities reported in the news remind us that many of these devices have been designed without security in mind. Nowadays, typical PC systems are hardened against common software vulnerabilities. Unfortunately, this is not the case for most embedded systems. For instance, in a PC, process separation is achieved through virtual memory, protection against stack and heap based buffer overflows are commonly inserted by compilers, and exploit mitigation such as Address Space Layout Randomization (ASLR) is adopted by most operating systems. In addition, static analysis techniques for executable code have greatly evolved in the last ten years. For example, Clang's [3] static analysis is now able to catch many common bugs (e.g., some buffer overflows) at compile time. While not perfect, these countermeasures make traditional systems more resilient against attacks. Meanwhile, compilers used to produce software for embedded devices (firmware) often lack such protection mechanisms. Runtime exploit mitigation mechanisms such as ASLR or Data Execution Prevention (DEP) are not present or provide only a fraction of the protection offered by their PC counterparts. Moreover, many countermeasures are often omitted due to constrained budgets, limited hardware resources, or lack of incentives.

Nevertheless, these systems are often connected to the Internet and exposed to the same security threats as traditional server applications [8].

In this paper we focus on locating complex code that is driven by user input. The most common examples in this category are parsers, but we generically refer to such code as PARC$_3$ (PArser-like Routines and Complex Control-flow Code). In practice, parser components of embedded devices represent the first line of defense in charge of processing and decoding external input. The fact that they are directly exposed to possibly malicious or malformed data makes parsers critical from a security perspective [18, 12]. In addition, parsers are often implemented using complex routines and string manipulations that are themselves prone to security bugs [34].

Complex code and parser definition

In this paper, we refer to input parsers in a loose sense. Distinguishing between a lexer to separate the input stream in logical tokens, and a parser to transform the token stream into an abstract syntax tree, as is customary in compiler literature is neither relevant nor practical for our purposes. In the binary the two stages are often inseparable, due to macro expansion and inlining performed during the compilation process of the firmware. Moreover, there might not have been separate stages for lexing and parsing in the software's

[*]VU University Amsterdam
[†]EURECOM
[‡]Radboud University

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA

ISBN 978-1-4503-3682-6/15/12...$15.00

DOI: http://dx.doi.org/10.1145/2818000.2818035

design in the first place. We want to identify $PARC_3$ components in firmware, i.e., *all* the code that processes hardware input, and takes control flow decisions based on this input. This includes, drivers, protocol parsers, and string tokenizers. In case the tokenizer and the actual parser are separated in the binary form, our approach would identify both as candidate parsers. As programs need complex control code that operates on external data for either activity, identifying such code implies the discovery of parsing code. For this reason, we use the more generic term $PARC_3$ rather than parser to refer to our target functions.

Analyzing the security of embedded devices

Even though it is typically feasible to extract the firmware of an embedded system (e.g., from a memory dump or an update file), it is often very difficult to perform an automated analysis of its code. First, firmware is almost invariably stripped of debugging symbols, and contain few strings – as many devices do not even have an interface to output text to users. Second, unlike applications for regular PCs, firmware images are mostly distributed as flash chip images. The hardware abstraction, operating system, application and data form a unity designed to work exactly on the hardware platform to which they are deployed. Without knowledge of the build process, it is hard for a security analyst to even locate functional units inside this blob. From our experience, even with access to the source code and knowledge of, say, a crash location inside the firmware, it is difficult to determine the root cause of the problem. Consequently, an independent security assessment on a binary firmware image is even more difficult and time-consuming. Finally, dynamic analysis of a firmware image is hard, as the dependence on the hardware is extremely tight. Unless one has a faithful emulator for the entire embedded device (which is rarely the case), or can debug the firmware on the device, dynamic analysis is not a feasible option. The firmware code needs the whole embedded system environment, consisting of peripherals that can be accessed via I/O memory regions, as well as hardware interrupts, to exhibit the same behavior as if it was running on the embedded device.

Analyzing $PARC_3$ functions in embedded devices

The security of parsers and $PARC_3$ code is a general problem and some of the techniques we present in this paper are not limited to embedded systems. However, several factors make the problem of analyzing $PARC_3$ code more difficult, and also more interesting, in embedded devices.

- In embedded systems, protocol stacks are often homegrown as vendors frequently reimplement even well-known protocols, such as Hypertext Transfer Protocol (HTTP) and Transmission Control Protocol (TCP), from scratch. They do so for a variety of reasons, e.g., resource constraints, licensing issues, and legacy considerations. Unfortunately, these implementations seldom undergo the same amount of scrutiny and security analysis as code that has been tried and tested in many other environments.
- The usage of lower-level programming languages like C and C++ is frequent in embedded systems. Many protocols are textual and therefore consist of text parsing, which is, surprisingly perhaps, still one of the most difficult operations to perform securely in those languages.
- The lack of clearly defined system APIs or kernel interfaces in embedded devices makes it challenging to locate the code that actually receives and processes input values. This is also made harder because of the lack of documentation and because the firmware is often monolithic. In a traditional operating system, such input is often provided by a system call or in an environment variable. On the contrary, in an embedded system it is often read from an unknown custom hardware I/O port.

In this work we therefore focus on detecting and analyzing implementations of such code in embedded systems, without the availability of the original source code or documentation.

There are many interesting use cases beyond its obvious offensive applications. For instance, we believe it is important to be able to perform third party security evaluations of embedded systems. However, manufacturers frequently do not have the incentives and resources to hire a third party. In many cases software components are integrated in larger systems (like a car) and a security evaluation may need to be performed on the code provided by a component supplier, who may not provide any assistance. Independent security analysis is also mandatory when the original manufacturer is not trusted. For example, a code that deals with users' input is a good place to hide 'features', such as undocumented commands, deviations from the protocol specifications, or even hard-coded backdoors [14].

To solve these problems, this paper presents a novel technique to automatically discover and analyze $PARC_3$ code on embedded devices, with the goal of detecting exploitable bugs, extract protocol specifications, and find hidden commands. The system we propose, named *PIE* for Parser-like code Identification in Embedded Systems, first translates firmware images from binary code to the Low-Level Virtual Machine (LLVM) compiler intermediate code. Based on a classifier for statically extracted features from this intermediate code, we can detect functions that contain parser components and complex code. The classifier is trained on code samples with known parser and decision code, e.g., the coreutils programs and several servers with complex protocols.

Contributions

In this paper we propose a novel analysis methodology to discover complex code related to parsers in embedded systems. Our technique is implemented in a prototype tool, which was successfully tested on several devices. To summarize:

- we present static analysis techniques for firmware, that relies on reverse-translation to LLVM, to perform generic detection of $PARC_3$ code,
- we demonstrate the effectiveness of our techniques by evaluating them on four real world devices, e.g., to extract all implemented commands from known protocol parsers (including "hidden" commands not specified in the manual), and/or detect memory related bugs in input handling code.

We intend to release all source code of *PIE* to the public, to be available to the research community as a base for further development.

2. RELATED WORK

In this section we summarize related projects and provide the basis of the analysis methods we used in our approach.

Static binary analysis and machine code translation
To deal with the complexity of machine instruction sets, firmware analysis is often performed by translation of the binary opcodes into a intermediate language which explicitly express all side effects of the machine instructions. Notable binary analysis frameworks are the Binary Analysis Platfom (BAP) [13] and its predecessor BitBlaze [42]. However, support for non x86 architectures is limited and fixing or extending these framework would require a considerable amount of engineering effort. LLVM has been previously integrated in cross-platform dynamic analysis systems such as S^2E [21] and Panda [25, 44]. We therefore decided to translate our program to the LLVM [31] intermediate language, on which we then perform our analysis. While LLVM was designed as a compiler intermediate language, its simplicity and the availability of various transformations makes it an excellent target for decompilation. For the Intel x86 instruction set, there are several translators to LLVM [20, 10, 7]. Our translator is derived from RevGen [20], which now has been incorporated into S^2E [21]. Various transformations and analyses have been implemented for the LLVM intermediate language, including static slicing [41] (introduced by Weiser [43]) and integer range analysis (proposed by Navas et al. [33]).

Symbolic and concolic exploration of binary code
Symbolic execution is a technique that was first proposed in 1976 [30]. Since then, many symbolic execution systems have been developed, including S^2E, KLEE, FuzzBALL, and JPF. However, to the best of our knowledge, S^2E is the only one which can target the ARM architecture.

Selective Symbolic Execution (S^2E) is a framework developed at EPFL that allows symbolic execution of binary code. It leverages QEMU [9] to translate blocks of binary instructions to an intermediate language, which in turn are translated to LLVM [31] instructions. If symbolic values are touched by the instruction block, KLEE [16] then executes the LLVM code symbolically. Using the plugin interface of S^2E, one can hook into instruction translation, execution, memory access and various other events.

Concolic execution [26, 38] is an optimization of symbolic execution. It uses a pair of a concrete input and a symbolic variable to represent a concolic value. We use concoilc execution instead of taint tracking because concolic execution has the advantage of tracking the full data-flow history instead of a short summary encapsulated within the tainted variable. This extra information can be used to make more precise inference about the tracked data.

Dynamic analysis of embedded devices
Avatar [45] is an open source solution to perform binary analysis on embedded systems' firmware. It executes binary code in an instrumented emulator, but avoids emulating the whole system by forwarding I/O accesses to the embedded device, where peripheral accesses are performed. In particular, the execution of a firmware in an emulator allows us to trace all executed instructions and memory accesses.

In Firmalice [40], Shoshitaishvili et al. use a mixed approach of static and manual analysis to identify authentication bypass backdoors in firmware. Points in the control flow which can only be reached when a user is authenticated are identified semi-automatically. The framework then assists the analyst in finding control flows which reach this point from an unauthenticated state without proper authentication (i.e., through hidden commands or hard coded credentials). Our work, while using similar techniques, aims at providing a more automated way of parser detection and has different goals. We aim to identify code that performs parsing in general, and not only code that is related with authentication.

Protocol learning
Polyglot [15] differs from previous work on protocol reverse engineering in that it proposes a technique called shadowing to extract protocol specifications from a program binary. By observing how the program interprets received messages, the system is able to identify fixed length fields, variable length fields and keywords. The same approach of white-box execution analysis is followed by Tupni [23] to reverse binary file formats. In addition, it can use information from several example input files to gain more accurate information on file fields. Prospex [22] identifies similar protocol messages and clusters them to recover the protocol's state machine.

Automatic reverse engineering
RevNIC [19] is a tool to automate reverse-engineering of device drivers. The authors demonstrate on the example of a Windows network driver that RevNIC can use symbolic execution to explore the device driver's code, slice instructions related to the driver, and build a synthesized driver from the extracted hardware model. SymDrive [36] uses a very similar technique of exercising drivers with symbolic execution. The focus of this work is to find bugs in operating system drivers, without the need of the physical device that the driver is developed for.

Code complexity and embedded parsers
Code complexity metrics have been used by Pan et al. [35] for bug classification and detection at the source code level. Cyclomatic complexity is a metric introduced by McCabe [32] that measures code complexity. However, Shin and Williams [39] suggests that the correlation between source code bugs and cyclomatic complexity is insignificant. New metrics have to be used if the goal is security. Research done by Chen et al. [18] shows that implementing embedded interpreters significantly increases the attack surface of systems. The authors provide a classification of common bugs occurring in embedded interpreters, which for example include difficulties to implement interpreters correctly in unsafe languages, incorrect handling of arithmetic errors, and preventing resource exhaustion and arbitrary execution.

3. STATIC PROGRAM ANALYSIS

The goal of the static analysis performed by *PIE* is to identify $PARC_3$-like parts inside firmware code—i.e., routines associated with the analysis and parsing of data. The most interesting examples of such code are parsers. We first studied parsers to identify common features. As we shall see, features based not only on control flow, but also on data flow are stronger to successfully tell such code from other code.

3.1 Identification of parser characteristics

In this paper we use the term $PARC_3$ to describe any piece of code dedicated to consume external input and either build

an internal data structure for later use, or orchestrate the execution of the proper functionality based on the input values. Such code is often referred to simply as a parser and has been extensively studied in the compiler community [6] as a way to perform a syntactic analysis of a computer language. For a deeper understanding, we will now specifically analyze parsing in a little more detail. In practice, the distinction between lexing and processing is not always clear and the general structure of a parser can become quite difficult to model. For instance, parsers in embedded systems are often hand-written and do not strictly separate between the two stages. Even worse, often part of the software behavior begins execution before the entire input is parsed, leading to undefined internal states if the remaining input does not correspond to what was expected [11, 37].

As mentioned earlier, the distinction between lexers and parsers is not interesting for this paper. Similarly, we are not interested in the details of the parser algorithms (e.g., top-down or bottom-up). As long as they have a token-fetching loop and an internal state machine, *PIE* will recognize them.

To get an understanding of what a typical parser looks like in binary format, we built a dataset containing several examples of parsers built with `LEX` and `YACC` [29], as well as custom parsers from open-source firmware. We then compiled them to `ARM` machine code, and reverse translated the ARM instructions into the `LLVM` intermediate language as described in Section 3.2. We use an intermediate representation to perform both control and data flow analysis in a format independent from the underlying machine language.

A common characteristic we noticed in these examples was the presence of two distinct patterns. The first pattern consists of the loop where the input data is fetched (e.g., by processing characters of a string or retrieving stream data from a device). The second recurrent pattern comprises the decision code, which often contains many conditional branches that depend on input values. Unfortunately, it is hard to generalize these findings. Also, not all the parsers expressed these two patterns clearly. For example, a parser might be event-driven and called by an interrupt handler to process the next character, or the decision control flow might be spread over several functions—making it difficult to detect in an automated fashion.

Since it is hard to propose general rules, we decided to extract a number of simple features and use machine learning to identify code that likely belongs to parsing routines. Each feature measures certain aspects of the code. For PARC_3 code, we are interested in code complexity, as well as in the way in which certain values influence control flow. These features are weighted and then combined to a single scalar value, which is an indicator of the function's likeliness to contain a parser.

3.2 Lifting to an intermediate language

Direct static analysis of assembler code is hard because instructions tend to have side-effects. Thus tracking data flow across assembler instructions is non-trivial. For this reason, we chose to translate the machine code to an intermediate language where instructions are side-effect free.

The LLVM intermediate language is well-suited for our purpose as data and control flow are easy to extract from its Static Single Assignment (SSA) representation. Further, using a common intermediate language instead of a particular machine language makes it easier to reuse developed techniques across different instruction set architectures. State of the art frameworks for translating machine code to LLVM did not fit our purpose as they either do not support ARM [10, 24] or have only partial support for it [17].

Because S^2E uses LLVM internally and is able to run ARM code, we chose it to perform the translation from machine code to LLVM. Since S^2E is originally implemented as a dynamic analysis framework, which translates code on the fly, our plugin had to significantly alter the operating principle of S^2E. Our solution was to use the translation functionality to progressively translate the binary, one basic block at a time, without executing the resulting code. By analyzing the recovered code, we can discover new basic blocks, similar to the process performed by a recursive disassembler.

Because the generated LLVM code still retains many of the constructs of the original machine language, we apply a set of transformations to normalize the results, bringing it closer to a compile-time representation. The following transformations are implemented partly as passes which are run using LLVM's "opt" utility, and partly as python scripts using `llvmpy` [4] to inspect and manipulate LLVM code.

Control Flow Normalization. To obtain a useful control flow representation, a function recovery pass connects translated LLVM basic blocks and groups them into functions. Functions are detected based on call and return patterns. Further, jump table patterns are detected and transformed to switch statements. This transformation is very important, as switch statements are recurring patterns in state machine implementations, which are often used in parsers. Optionally, in this step, we make use of information provided by external dissasemblers.

Data Flow Normalization. Data flow in SSA form is considerably easier to programmatically follow than data stored in global values or stack memory. This is why we wanted to convert accesses to the assembler stack and global variables to SSA form whenever possible. In a first step, we replace accesses to Qemu's internal representation of program memory to normal LLVM load and store instructions. A second pass detects memory accesses relative to the assembler stack pointer and transforms them to SSA. This pass first analyzes the assembler stack usage (by tracking the value of the stack pointer across the function), and then creates new SSA variables for every stack location referenced with a constant offset from the stack pointer inside the function.

Finally, we apply the `scalarrepl` standard LLVM pass, which breaks the structure data type created for the stack frame into individual variables, and the `mem2reg` pass, which transforms local variables to SSA form.

3.3 Features of PARC_3 components

For the actual detection of complex and parsing code in a firmware, we extracted a set of features from the control flow graph (CFG) and data flow graph (DFG) of each function.

Looped `switch` *statement (*`switch_loop`*)*

Sequential parsers are typically implemented as a state machine. New tokens are fetched in a loop, and the next state or action is decided based on the current state and the next input token. This decision process usually involves switch statements or dispatch tables. Thus, identifying loops with

switch statements or dispatch tables in their body are a good indicator for parsers, especially if the value influencing control flow inside the loop's body in turn depends on the loop's induction variable.

Data flow analysis on conditional statements (`br_fact`*)*

While detection of switch statements already covers a large portion of parsers, compilers can choose to lower switch statements to conditional branches, or hand-coded parsers use conditional statements for example in conjunction with the `strcmp` function.

For this reason we analyze the influence of each variable on control flow decisions, yielding a "branching factor". The branching factor is computed by first assigning all instructions the number 0. Then, we iterate over all conditional instructions in the function (*branch*, *select* and *switch* instructions). We perform a simple recursive data flow analysis on the condition value, and add the number of outgoing edges from the conditional instruction to each instruction's number. In the end we pick the highest branching number to represent the maximum branching factor for a single value in that function.

Maximum number of incident edges (`in_edges`*)*

Usually parsers are implemented as a sequential token processing loop. Control flow paths fan out in the loop's body, and rejoin in the loop head basic block. Thus, a node with a lot of incoming edges is more likely to belong to complex parsing code, and the number of incoming edges then reflects the number of different execution paths.

Number of basic blocks (`bb_cnt`*)*

This feature simply reports the number of basic blocks in a function. The rationale is that embedded code, when written and compiled for minimum size, often implements parsers in a single huge assembler function. Smaller functions that may be present in the source are often inlined.

Number of callers (`call_cnt`*)*

Some tokenizing functions (like *atoi* or *scanf*) are called from a lot of program locations. Given this observation, we use the number of callers as a feature. On the one hand, frequently-called parsing functions can reveal further details when instrumented in a dynamic analysis. On the other hand, a programming error in an often-called function can be exploited in many parts of the program.

Switch statement (`switch`*)*

Complex input handling code often uses switch statements and jump tables as a way to divert control flow to appropriate handling code. For this we take into account the "switch" statement and jump tables. Even though this metric overlaps with the `switch_loop`, the evaluation shows the importance of this metric.

4. TRAINING AND EVALUATION

In this section we describe how we determine the performance and relative weight of the heuristics proposed in Section 3.3 using regular Linux applications. For our analysis, we first obtain the LLVM bitcode files with complete control flow information (by compiling popular open source software with `Clang`) for a data set consisting of 101 coreutils and 3 popular applications (ProFTPd, lighttpd, and bash). We manually inspected all the coreutils and labeled each target function accordingly. For the other applications, we sample the program to mark some functions as "parser-like" (as a sanity check for obvious false negatives), and exhaustively check the result returned by *PIE* for false positives.

4.1 Scoring

As a combined score for the heuristics we normalize and weight the individual scores as follows

$$score = \sum_{f \in features} \omega_f \frac{x_f - min(X_f)}{max(X_f) - min(X_f)} \qquad (1)$$

where x_f represents the value feature f takes for a given function, X_f is the set of values that f takes for all tested functions, and ω_f is the weight attributed to feature f. Thus each feature is normalized to a value between 0 and 1, and the total score is bounded. The goal of this section is to find the appropriate weights and threshold T such that if the score of a particular routine exceeds the threshold, we can declare it to be PARC_3 code. False positives (FPs) are functions that score above the threshold but are not PARC_3 code (according to manual analysis), while false negatives (FNs) occurs when PARC_3 functions score less than T.

4.2 Validation

We apply 2-fold cross-validation by splitting our data set in two subsets: S_0 and S_1. We perform training by running *PIE* on S_0, and do the validation on S_1 (later we also swap the sets). We then test each possible ω_f and T combination (in 0.1 increments), and for each parameter combination, we compute the minimum distance on the ROC graph from the ROC curve to the optimum ($FP = 0$ and $TP = 1$). The output of the training step is a set of ω_f and T parameters, ordered by the distance from the optimum. To obtain the weights (ω_f), we average the best $K\%$ results from the training step. We then compute the average of FP and TP for the validation set S_1 using the output of the training step.

4.3 Cross validation results

Table 1 shows the results of the cross-validation. The first row shows results with training on S_0 and validation on S_1 ($S_0 \rightarrow S_1$). The second row ($S_1 \rightarrow S_0$) shows cross-validation when the two sets are swapped. We display the output parameters, the threshold T and the weights as follows: `switch_loop` ω_0, `br_fact` ω_1, `in_edges` ω_2, `bb_cnt` ω_3, `call_cnt` ω_4 and `switch` ω_5. We compute FP and TP rates using the parameters gauged in the training step. We also display the distance D from the $< FP, TP >$ point to the optimal. Because we define our goal to be as close as possible to the optimal point, D is used to estimate the error ($\varepsilon = |D_{S_0 \rightarrow S_1} - D_{S_1 \rightarrow S_0}|$) of our method. Figure 1 shows the optimal values of the parameters using the best K% samples. Figure 2 displays example ROC graphs for ω_f, corresponding to $K = 2\%$. The figures show that finding good, stable values for ω_f is straightforward and produces very good ROC curves.

5. CASE STUDIES

To show the use of *PIE* on real word embedded devices, we applied the tool to four case studies: a GPS receiver, a power meter, a hard disk drive, and a PLC. All four devices

K	Direction	Training							Validation			ε
		T	ω_0	ω_1	ω_2	ω_3	ω_4	ω_5	FP	TP	D	
0.5%	$S_0 \to S_1$	0.234	0.708	0.483	0.278	0.349	0.123	0.691	0.032	0.982	0.037	0.0178
	$S_1 \to S_0$	0.218	0.719	0.566	0.295	0.473	0.119	0.675	0.016	0.990	0.019	
1%	$S_0 \to S_1$	0.244	0.696	0.496	0.363	0.353	0.122	0.689	0.041	0.996	0.041	0.0223
	$S_1 \to S_0$	0.220	0.709	0.545	0.300	0.482	0.122	0.691	0.016	0.990	0.019	
2%	$S_0 \to S_1$	0.247	0.695	0.502	0.364	0.413	0.134	0.685	0.041	0.996	0.041	0.0223
	$S_1 \to S_0$	0.221	0.704	0.534	0.340	0.480	0.123	0.685	0.016	0.990	0.019	
5%	$S_0 \to S_1$	0.263	0.687	0.502	0.412	0.515	0.138	0.670	0.032	0.987	0.035	0.0122
	$S_1 \to S_0$	0.234	0.686	0.521	0.453	0.511	0.129	0.673	0.020	0.990	0.022	
10%	$S_0 \to S_1$	0.282	0.642	0.512	0.443	0.547	0.161	0.625	0.034	0.964	0.050	0.0272
	$S_1 \to S_0$	0.239	0.618	0.514	0.529	0.509	0.134	0.605	0.020	0.990	0.022	

Table 1: Validation for $K = \{0.5, 1, 2, 5, 10\}\%$.

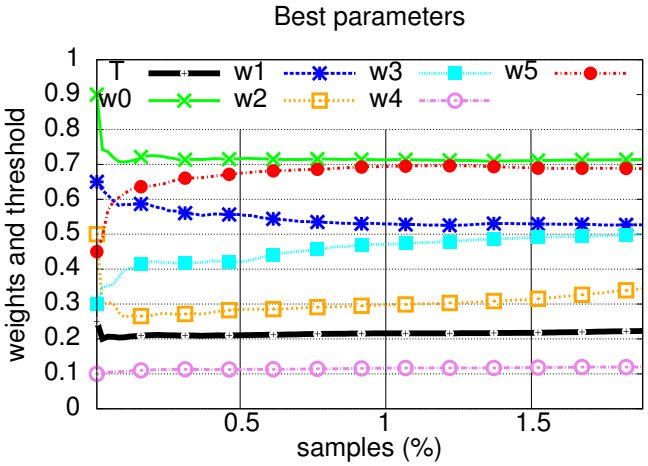

Figure 1: Best parameters for the complete data set.

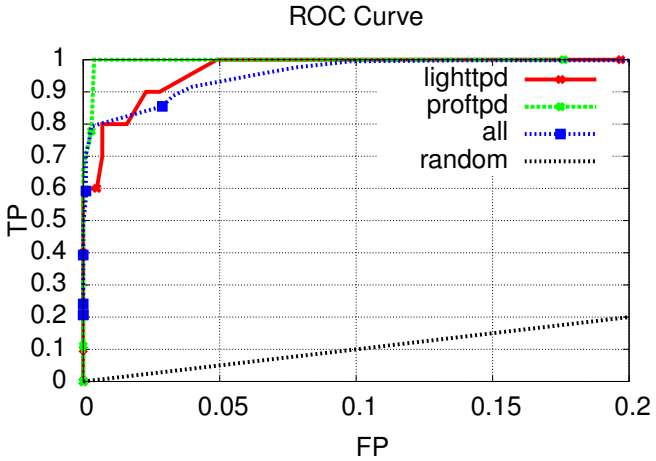

Figure 2: ROC plot using ω_f corresponding to $K = 2\%$

	GPS	Meter	HDD	PLC
Size (kB)	519	544	287	9,984
# Basic blocks	48,738	55,054	43,866	657,349
# Functions	1,098	2,332	6,338	15,437
# CG edges	1,299	3,373	3,152	24,945

Table 2: Firmware sizes.

use SoCs or MCUs that rely on an ARM CPU core. Table 2 shows the complexity of their firmware in terms of number of basic blocks, functions, and call graph (CG) edges, as counted on the output of the LLVM translator. For all test cases we use *PIE* to select interesting (PARC_3) functions and briefly discuss them. For the HDD and the PLC we use the output of *PIE* as a basis for starting a more advanced analysis to demonstrate *PIE*'s usefulness from the security perspective.

5.1 GPS receiver

In the first experiment we analyzed the firmware of a USB GPS receiver stick. The device has a "boot loader mode", where it receives a binary over an emulated serial port interface on the USB connection, and subsequently executes it. Using SiRFDemo and SirFFlash utilities one can interact with the device and read and update the firmware. This first experiment intends to show that *PIE* is effective in selecting parser related functions and complex functions.

PIE results. For the values corresponding to $K = 2\%$ and $T = 0.247$, 2.3% of functions were marked as PARC_3 code and the false positive rate is 0.047. Apart from common `sprintf` and `scanf` functions which were correctly recognized as containing an input parser, *PIE* found a function which applies the Viterbi algorithm. Viterbi algorithm is used for signal processing to decode noisy signals and consists of complex code. Loosely, it can be viewed as a parser of the data provided by the GPS receiver. Another function automatically detected by our system parses data which seems to be in the Motorola SREC format. This is to be expected as the device's updates are in SREC format. *PIE* discovered parts of the SiRF III protocol as well which was confirmed by reverse engineering. The SiRF protocol is a binary protocol, for which simple heuristics like strings search are not effective.

5.2 Power meter

In our second experiment we tested a remotely controlled electric energy metering device, also commonly referred to as a *smart meter*. The power meter contains, among other components, a GSM/GPRS modem and an infrared interface used to program and calibrate the meter.

PIE results. The false positve rate is 0.046 and 3.79% of functions were marked as PARC_3 code. Our tool automatically identified several functions responsible for string pro-

cessing, similar to `scanf` and `sprintf`. In all cases, the format specifier is parsed in order to know what type of data needs to be printed. *PIE* also found two parsers implemented with a switch table over constant printable characters. We believe that this is the function that deals with the infrared interface of the power meter, since the switch seems to correspond to the ISO-IEC62056-21 protocol—mentioned in the manual of the device. *PIE* also identified the GSM/GPRS modem command handler.

5.3 HDD

In our third experiment, we analyzed a commercial off-the-shelf ARM-based hard drive. After a discussion *PIE*'s results, we show how the output of our tool can be used to perform dynamic analysis to recover hidden commands in one of the detected parsers, and show surprising results.

PIE results. The false positive rate is 0.197 and 1.4% of functions were marked as $PARC_3$ code. We do not consider optimized versions of the `mem*` functions as $PARC_3$ code but *PIE* does so. The firmware has multiple versions of these functions, hence the higher false positive rate.

On the HDD, *PIE* found six core parsing functions among the ten highest scoring functions. In particular, the two functions with the highest score (and well above the threshold T) are the parsers for a simple UART menu used for maintenance, and the parser for receiving iHex-formatted firmware updates. Interestingly, the advanced UART menu (which can be enabled from the maintenance interface) was not discovered by our system. It turns out that since each character is treated on a different invocation of the UART receive interrupt and processed directly, the state machine of the menu is distributed over several functions. The experiment shows that, while the output of the *PIE* contains false positives (in this case arithmetic functions) and false negatives, an analyst can quickly identify the main parsers by looking at the functions with the highest score. Considering that the entire firmware contains over 6300 functions, our tool considerably reduces the amount of manual analysis time.

Example of dynamic analysis: ATA command parser

As an example of how we can use *PIE* for deep analysis, we analyzed one of the parsing functions we found, the AT Attachment (ATA) command parser (the hard drive's interface to the computer). The ATA protocol is a simple command-response protocol, where a fixed structure containing block address, access size, device, etc. is sent to the hard drive. The protocol is particularly interesting to analyze because numerous commands have been changed or deprecated since its first specification in 1994, and it is well known that vendors often implement custom commands.

We wanted to know which commands our drive supports and especially if there were any commands not described in the ATA specification. Our idea was to use concolic execution on the ATA command parser, and to discover implemented commands and command options in this fashion.

For dynamic analysis, we used the *Avatar* [45] framework. We first executed the firmware, with an injected debugger stub, on the HDD. Once *PIE* detects a $PARC_3$ point of interest, we pause execution on the device with a breakpoint and take a full snapshot of the HDD's memory and resume execution from that snapshot in S^2E. Accesses to I/O memory ranges are forwarded to the HDD (which is still stopped at the breakpoint), and also recorded in a trace. With the memory snapshot, and the recorded trace, we can now replay an execution. During the replay, we make use of S^2E's symbolic execution engine to explore the impact of different inputs on the parser.

We used an S^2E plugin to re-execute the recorded program path and identify commands and options by marking them as symbolic values. Whenever the execution left the pre-recorded execution path, we removed the symbolic state and noted the newly discovered values to inspect them later (similar to SAGE [27]).

This way, we successfully identified 78 ATA commands, most of which are documented in the standard [1]. Interestingly, some commands specified in the standard were not implemented by our disk, like "Read Direct Memory Access (DMA) Queued EXT" (`0x26`). On the other hand, the drive implements some commands that are marked as obsolete, retired or vendor specific. Among the undocumented vendor specific commands, two are particularly interesting:

- `0x80` looks like a gateway for internal firmware commands. Sending this opcode with all other registers set to "0" corrupted our drive's configuration to the point where we needed to re-flash the firmware.
- `0xEA` checks for a magic logical block addressing (LBA) of 0x333324 ("$33"), and sets a configuration value if this constant is set.

The other three vendor specific commands, `0xFA`, `0xFC` and `0xFD`, seem to be related to normal hard drive operations. During our analysis we made another interesting observation concerning the ATA NOP (`0x00`) command. If the special constant `0x7654321` is presented as LBA, a second register value is treated as sub-opcode, and another parser is invoked. Depending on this sub-opcode, several functions can be called, including firmware update functionality. As the foundation of all of our analysis and results was rooted in *PIE*, we believe the experiment shows that *PIE* is useful as a starting point for the analysis of embedded firmware.

5.4 PLC

Our last case study concerns a PLC. Such a device is part of a Supervisory Control And Data Acquisition (SCADA) infrastructure and it is normally used to automate processes in a factory. PLCs are often embedded systems that can receive inputs from sensors, send outputs to drive actuators (e.g., motors or valves), and which are equipped with a network or field bus connection to communicate with other systems in the infrastructure. Analyzing the security of this type of device is especially interesting, as they are used inside several "critical infrastructure" fields, such as power generation, water supply, and traffic control.

The PLC had the biggest firmware with the widest range of functionality among the four firmware we analyzed. It contains a proprietary operating system, the virtual machine for interpreting ladder logic programs, a web server with OpenSSL running on top of a TCP stack, and a Remote Procedure Call (RPC) library to communicate with other SCADA components and the computer used to program the PLC). The web server and the proprietary control protocol parser are two particularly interesting targets for attackers, as they are exposed over the network.

PIE results

With the treshold $T = 0.247$ we do not obtain any false positive but we miss many of $PARC_3$ like functions. The firmware size of the PLC is one order of magnitude bigger than the firmware of GPS, the one of power meter, or than the samples in the training set. The accuracy of our training for T is biased towards small firmware. However, the *score* is still usable: the false positive rate for the top 0.5% functions sorted by *score* is 0.023. Among the top 0.5% functions, we notice code belonging to the OpenSSL library. OpenSSL is notorious for its complex code and for the large amount of parsing operations.

As an example we list some of the functions with $PARC_3$ functionality identified by *PIE*:

- `calls_OMSp_serializer_parser_parser`[1] (position 4 in the PIE ranking) – this function is part of the *ISO-TSAP* protocol of the *OMS* proprietary storage format. From this point it is easy to detect all the other handlers for these protocols. The implementation the *ISO-TSAP* has been a source of errors in the last versions of this firmware.
- `miniweb_source_MWEB_VarWriter`[1] (position 5) – this function resides in the web server module and it writes values to a predefined variable in the main application. The function contains a parser for the name of the variable, the value of the variable, and the type of the variable.
- `firmware_update_check`[2] (position 21) – this function checks the file format of the firmware update. A firmware update file can be uploaded via the web interface of the PLC or via a special MMC card.
- `internal_var_print`[2] (position 36) – function that formats a string. The type specifier tokens are not common. We believe that this function is used to print PLC's internal variables in file logs.
- `recurisve_path_lexer`[2] (position 55) – a function used by the PLC's web server that symplifies URIs. We believe that this function is used for parameter tokenization of HTTP requests. This function is a perfect example of $PARC_3$ code that can hide bugs: it is large (756 basic blocks) and it is recursive.
- `boot_menu`[2] (position 58) – as we shall see, this is an undocumented feature providing a hidden boot loader menu.

Finally, we chose two points of interest for further study: a parser in the boot loader accepting commands over the serial port, and the Uniform Resource Locator (URL) handling of the embedded web server. Our motivation for analyzing the boot loader parser was to understand its purpose and find a way to inject a debugging code stub in the system without hardware intervention. The web server is obviously an interesting target, as it is reachable over the network. Moreover, a quick Shodan [5] search reveals many PLCs which are directly exposed to the Internet.

[1]Function named after corresponding error messages.

[2]Function does not refer to any error message. The name is given by functionality.

Example of dynamic analysis: boot loader parser

A quick view of the code leading to the boot loader parser showed that it is only activated by a special sequence of bytes. To extract this sequence, we used symbolic execution to injected symbolic bytes whenever the serial port was read. By looking at the values' constraints in the symbolic state where execution enters the parser, we directly obtained the activation string. Subsequently, we sent the string to the PLC's serial port, and the boot loader dropped in a command-response mode. Using the same technique to inject symbolic bytes when the serial port is read, we were able to understand the binary message format. Each message is prefixed with a length field and an opcode field, followed by a payload. The last byte is a simple checksum.

By observing the triggered code and the replies from the boot loader, we could also understand the meaning of messages. There were messages for querying the hardware and boot loader version whose purpose was obvious from the reply. The other messages all trigger accesses to different peripherals of the PLC, and allow for example to toggle the LEDs used to display the PLC's status.

Thus we assume that the boot loader command interpreter is used to test the hardware without the full firmware. However, we were not able to identify a command which would allow us to read and write arbitrary memory.

Example of dynamic analysis: HTTP request handle

The PLC's web server is custom and serves a mix of static content, processed templates and internal values. By default, it allows starting and stopping the process' execution as well as inspecting and modifying program variables, input and output values. If the engineer chooses to, he can also embed "user pages" in his process which show the process' state and permit control in a visually pleasing way.

We wanted to focus on the parsing of URLs, as most data is sent to the web server via HTTP GET requests. Using the output of *PIE*, we quickly identified the handler for the GET request. Starting from a snapshot taken at the beginning of the parser function, we replaced the URL with a short string of symbolic values and continued execution in S^2E.

We found that the parser function returns an error code, which conveniently tells us if the symbolic URL was accepted by the request handler or not. Leveraging this information, we could focus on symbolic states where the URL was accepted, which revealed some interesting parameters. For example, we found that by inserting "`?SRC`" in the URL of a dynamically generated page, the web server would return the page template's source code. Most probably this parameter was used to debug web page templates or the template engine, but it is highly questionable if such undocumented parameters should be present in a release version of the PLC.

Second, symbolic execution revealed a URL prefix which exposes a web service Application Programming Interface (API). Based on the URL, we suspect that this API might be used for controlling a PLC via an IPhone. While some of the services exposed require authentication, undocumented interfaces in a security critical device should be seen with caution. Other vendors have been known to implement hidden APIs exposing privileged operations "just for convenience" [28].

Third and last, we found a bug, resulting in a software crash, in one of the input parameters. A pseudocode representation of the vulnerable code can be seen in Listing 1.

`strtol` parses the hexadecimal number from the parameter, and returns it as a 32-bit value. Afterwards the function proceeds to check that the parsed number does not exceed a maximum threshold, and then loads a pointer from an array of structures. Even though the function seems secure, as the maximum value of the parsed number is checked, it is not: `idx` is a signed value and can be negative. This negative index is then used later to access a structure, which causes a processor exception if no memory is mapped at the pointed address. This bug has been reported to the vendor and it is fixed in the most recent firmware version.

Listing 1: Negative index used in a table.

```
void get_from_hex(char *buff_in, void **ret)
{
 signed int idx; void *result;

 idx = strtol(buff_in, NULL, 16);

 if ( idx >= 50 ) result = NULL;
  else result = 76 * idx + 0xb44fdc;
 *ret = result;
}
```

We conclude that finding two backdoors and one bug on the PLC with the help of *PIE* shows that *PIE* is very useful for analysis from a security point of view.

6. FUTURE WORK

PIE would benefit from a more accurate data flow analysis algorithm. Combining data flow and template matching proved to be powerful, even with the limited data flow analysis that we implemented in our prototype. Using points-to analysis and inter-procedural analysis could result in a more complete picture of the code that is analyzed. Furthermore, detecting loop indices with data flow would improve the detection of lookup tables used by parser.

The next logical step to extend our system is to fully automate the application phase. As an example, it is possible to use *PIE* to automatically generate white-box fuzzing test cases, similar to SAGE [27] or AFL [2]. This would provide a fully automated system for the testing of a firmware.

We also believe that testing of embedded devices would benefit from software emulation of device peripherals. In this case, the problem is that there is no behavioral specification for a peripheral, making the process of modeling it in software a daunting reverse engineering task. This lack of specifications is reflected as well on the way the protocols are implemented. If machine-readable specifications for each protocol or feature in an embedded device were to exist, automatic validation would be a powerful tool. In this case, *PIE* could be extended to perform automatic validation not only for protocols but as well for the behavior of the device.

A better detection of error code paths would help *PIE* to provide more assistance in successive symbolic execution. While detecting error paths in normal binaries is easy (e.g., a segmentation fault is easy to detect in an operating system), detecting faulty states in embedded systems is non-trivial. Each firmware treats errors individually, which is why a more sophisticated static analysis has to be devised.

7. CONCLUSION

In this paper we described a new method for analyzing parser-like binary code in embedded devices, which we implemented in *PIE*. We established simple yet effective features to detect parsers and complex handling code, and evaluated them to show their potential for parser detection in binaries. We then demonstrated the practical impact of our work on four different embedded devices. For each device, we could detect complex and custom designed parsers, greatly reducing the required manual analysis time. Our case studies show our techniques can help to address the urgent problem that we currently lack almost all knowledge about unknown protocols, hidden interfaces, and additional unspecified functionality in embedded devices. We hope that improving awareness and third party analysis will help improve trust in such devices.

Acknowledgments

This research was supported by the NWO CYBSEC OpenSesame project (628.001.006), the ERC StG Rosetta project, and by the European Union Seventh Framework Programme (project FP7-SEC-285477-CRISALIS). We thank our collegue Remco Vermeulen who provided insights about the cross-validation and about the training steps. We also thank our anonymous reviewers for their invaluable feedback.

8. REFERENCES

[1] Ata-6 protocol spec. http://t13.org/Documents/UploadedDocuments/project/d1410r3b-ATA-ATAPI-6.pdf.

[2] american fuzzy lop. http://lcamtuf.coredump.cx/afl/, February 2015.

[3] The Clang compiler. http://clang.llvm.org/, February 2015.

[4] LLPython website. http://www.llvmpy.org/, February 2015.

[5] Shodan search engine. http://www.shodanhq.com/, February 2015.

[6] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. 1986.

[7] Anand, K., Smithson, M., Elwazeer, K., Kotha, A., Gruen, J., Giles, N., and Barua, R. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. EuroSys '13.

[8] Atzori, L., Iera, A., and Morabito, G. The internet of things: A survey. *Computer networks 54*, 15 (2010), 2787–2805.

[9] Bellard, F. QEMU, a Fast and Portable Dynamic Translator. ATC '05.

[10] Bougacha, A., Aubey, G., Collet, P., Coudray, T., Salwan, J., and de la Vieuville, A. Dagger website, 2013. http://dagger.repzret.org/.

[11] Bratus, S., Locasto, M., Patterson, M. L., Sassaman, L., and Shubina, A. Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation. *;login: The USENIX Magazine 36*, 6 (Dec. 2011), 13–21.

[12] Bratus, S., Patterson, M., and Shubina, A. The Bugs We Have to Kill. *;login: The USENIX Magazine 40*, 4 (Aug. 2015).

[13] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. BAP: A Binary Analysis Platform. In *Computer Aided Verification*. 2011.

[14] Bunnie, X. Exploration and exploitation of an SD memory card, 30C3.

http://www.bunniestudios.com/blog/?p=3554, 2013.

[15] Caballero, J., Yin, H., Liang, Z., and Song, D. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. CCS '07.

[16] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI 2008.

[17] Carback, R. T. Fracture: Inverting the Target Independent Code Generator. Poster at the LLVM developer's conference, November 2013. https://github.com/draperlaboratory/fracture.

[18] Chen, H., Cutler, C., Kim, T., Mao, Y., Wang, X., Zeldovich, N., and Kaashoek, M. F. Security Bugs in Embedded Interpreters. APSys '13.

[19] Chipounov, V., and Candea, G. Reverse Engineering of Binary Device Drivers with RevNIC. EuroSys '10.

[20] Chipounov, V., and Candea, G. Enabling sophisticated analyses of x86 binaries with RevGen. DNS-W 2011.

[21] Chipounov, V., Kuznetsov, V., and Candea, G. *S2E: a platform for in-vivo multi-path analysis of software systems*, vol. 47. ACM, 2012.

[22] Comparetti, P., Wondracek, G., Kruegel, C., and Kirda, E. Prospex: Protocol Specification Extraction. In *Security and Privacy* (May 2009).

[23] Cui, W., Peinado, M., Chen, K., Wang, H. J., and Irun-Briz, L. Tupni: automatic reverse engineering of input formats. CCS '08.

[24] Dinaburg, A., and Ruef, A. Static Translation of X86 Instruction Semantics to LLVM with McSema. RECON 2014.

[25] Dolan-Gavitt, B., Leek, T., Hodosh, J., and Lee, W. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. CCS '13.

[26] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. PLDI '05.

[27] Godefroid, P., Levin, M. Y., and Molnar, D. SAGE: Whitebox Fuzzing for Security Testing. *Queue 10*, 1 (Jan. 2012).

[28] Heffner, C. Reverse Engineering a D-Link Backdoor, October 2013. http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/.

[29] Johnson, S. C. *Yacc: Yet another compiler-compiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[30] King, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[31] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. CGO 2004.

[32] McCabe, T. J. A complexity measure. *Software Engineering, IEEE Transactions on*, 4 (1976).

[33] Navas, J., Schachte, P., Søndergaard, H., and Stuckey, P. Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code. TOPLAS 2012.

[34] Newsham, T. Format string attacks. http://www.thenewsh.com/~newsham/format-string-attacks.pdf, 2000.

[35] Pan, K., Kim, S., and Whitehead, E. J. Bug classification using program slicing metrics. SCAM '06.

[36] Renzelmann, M. J., Kadav, A., and Swift, M. M. SymDrive: testing drivers without devices. OSDI '12.

[37] Sassaman, L., Patterson, M. L., Bratus, S., and Shubina, A. The Halting Problems of Network Stack Insecurity. *;login: The USENIX Magazine 36*, 6 (Dec. 2011), 22–32.

[38] Sen, K., Marinov, D., and Agha, G. CUTE: A Concolic Unit Testing Engine for C. ESEC/FSE-13.

[39] Shin, Y., and Williams, L. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on Quality of protection* (2008), ACM.

[40] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. NDSS 2015.

[41] Slaby, J. LLVMSlicer, Feb. 2015. https://github.com/jirislaby/LLVMSlicer/.

[42] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information Systems Security*, Lecture Notes in Computer Science. 2008.

[43] Weiser, M. Program Slicing. ICSE '81.

[44] Whelan, R., Leek, T., and Kaeli, D. Architecture-Independent Dynamic Information Flow Tracking. In *Compiler Construction*, vol. 7791 of *Lecture Notes in Computer Science*. 2013.

[45] Zaddach, J., Bruno, L., Francillon, A., and Balzarotti, D. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. NDSS 2014.