Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware

Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara
{yans,fish,christophe,chris,vigna}@cs.ucsb.edu

***Abstract*—Embedded devices have become ubiquitous, and they are used in a range of privacy-sensitive and security-critical applications. Most of these devices run proprietary software, and little documentation is available about the software's inner workings. In some cases, the cost of the hardware and protection mechanisms might make access to the devices themselves infeasible. Analyzing the software that is present in such environments is challenging, but necessary, if the risks associated with software bugs and vulnerabilities must be avoided. As a matter of fact, recent studies revealed the presence of backdoors in a number of embedded devices available on the market. In this paper, we present Firmalice, a binary analysis framework to support the analysis of firmware running on embedded devices. Firmalice builds on top of a symbolic execution engine, and techniques, such as program slicing, to increase its scalability. Furthermore, Firmalice utilizes a novel model of authentication bypass flaws, based on the attacker's ability to determine the required inputs to perform privileged operations. We evaluated Firmalice on the firmware of three commercially-available devices, and were able to detect authentication bypass backdoors in two of them. Additionally, Firmalice was able to determine that the backdoor in the third firmware sample was *not* exploitable by an attacker without knowledge of a set of unprivileged credentials.**

I. Introduction

Over the last few years, as the world has moved closer to realizing the idea of the *Internet of Things*, an increasing amount of the things with which we interact every day have been replaced with embedded devices. These include previously non-electronic devices, such as locks[1], lightswitches[2], and utility meters (such as electric meters and water meters)[3], as well as increasingly more complex and ubiquitous devices, such as network routers and printers. These embedded devices are present in almost every modern home, and their use is steadily increasing. A study conducted in 2011 noted that almost two thirds of US households have a wireless router, and the number was slated to steadily increase [22]. The same report noted that, in South Korea, Wi-Fi penetration had reached 80%. The numbers are similar for other classes of devices: it has been predicted that the market penetration of smart meters will hit 75% by 2016, and close to 100% by 2020.

These examples are far from inclusive, as other devices are becoming increasingly intelligent as well. Modern printers and cameras include complex social media functionality, "smart" televisions are increasingly including Internet-based entertainment options, and even previously-simple devices such as watches and glasses are being augmented with complex embedded components.

[1]For example, the Kwikset Kevo smart lock.

[2]Most popularly, Belkin's WeMO line.

[3]Such as the ION, a smart meter manufactured by Schneider Electric.

The increasingly-complex systems that drive these devices have one thing in common: they must all communicate to carry out their intended functionality. Smart TVs communicate with (and accept communication from) online media services, smart locks allow themselves to be unlocked by phones or keypads, digital cameras contact social media services, and smart meters communicate with the user's utility company. Such communication, along with other functionalities of the device, is handled by software (termed "firmware") embedded in the device.

Because these devices often receive privacy-sensitive information from their sensors (such as what a user is watching, or how much electricity they are using), or carry out a safety-critical function (such as actuators that lock the front door), errors in the devices' firmware, whether present due to an accidental mistake or purposeful malice, can have serious and varying implications in both the digital and physical world. For example, while a compromised smart meter might allow an attacker to determine a victim's daily routine or otherwise invade their privacy based on their energy usage, a compromised smart lock can permit unauthorized entry into a victim's home (or, in a corporate setting, a compromised badge access verifier can allow entry into extremely critical physical areas of an organization). In fact, this is not just a theoretical concern: there have already been examples of "smart-home" invasions [18].

Firmware is susceptible to a wide range of software errors. These include memory corruption flaws, command injection vulnerabilities, and application logic flaws. Memory corruption vulnerabilities in firmware have received some attention [12], [33], while other vulnerabilities have, as of yet, been relatively unexplored in the context of firmware.

One common error seen in firmware is a logic flaw called an *authentication bypass* or, less formally, a *backdoor*. An authentication bypass occurs when an error in the authentication routine of a device allows a user to perform actions for which they would otherwise need to know a set of credentials. In other cases, backdoors are deliberately inserted by the manufacturer to get access to deployed devices for maintenance and upgrade. As an example, an authentication bypass vulnerability on a smart meter can allow an attacker to view and, depending on the functionality of the smart meter, modify the recorded energy usage of a victim without having to know the proper username and password, which, is generally kept secret by the utility company. Similarly, in the case of a smart lock, an authentication bypass could allow an attacker to unlock a victim's front door without knowing their passcode.

Authentication bypass vulnerabilities are not just a theoretical problem: recently publicized vulnerabilities in embedded devices describe authentication bypass flaws present in several devices' firmware [15], [16], and a recent study has suggested that up to 80% of consumer wireless routers are vulnerable to *known* vulnerabilities [31]. In fact, an authentication bypass in Linksys routers was used by attackers to create a botnet out of 300,000 routers in February 2014 [6].

Detecting authentication bypasses in firmware is challenging for several reasons. To begin with, the devices in question are usually proprietary, and, therefore, the source code of the firmware is not available. While this is a problem common to analyzing binary software in general, firmware takes it one step further: firmware often takes the form of a single binary image that runs directly on the hardware of the device, without an underlying operating system[4]. Because of this, OS and library abstractions do not exist in some cases, and are non-standard or undocumented in others, and it is frequently unknown how to properly initialize the runtime environment of the firmware sample (or, even, at what offset to load the binary and at what address to begin execution). We term such firmware as "binary blob" firmware. These blobs can be very large and, therefore, any analysis tool must be able to handle such complex firmware. Additionally, embedded devices frequently require their firmware to be cryptographically signed by the manufacturer, making modification of the firmware on the device for analysis purposes infeasible.

These challenges make existing approaches infeasible for identifying logic flaws in firmware. Systems that are based on the instrumentation and execution monitoring of firmware on a real device [26], [33] would not be able to operate in this space, because they require access to and modification of the device in order to run custom software. In turn, this is made difficult by the closed nature (including the aforementioned cryptographic verification of firmware images) and the hardware disparity (any sort of on-device instrumentation would represent a per-device development effort) of embedded devices. Additionally, existing firmware analysis systems that take a purely symbolic approach (and, thus, do not require analyses to be run on the device itself) [12] would not be able to scale their analysis to complex firmware samples, like those used by printers and smart meters. Moreover, they require source code, which is typically not available for proprietary firmware. As a result of these challenges, most detections of authentication bypasses in firmware are done manually. This is problematic for two reasons. First, manual analysis is impractical given the raw number of different embedded devices on the market. Second, even when manual analysis is performed, the complexity of firmware code can introduce errors in the analysis.

To address the shortcomings of existing analysis approaches, we developed a system, called *Firmalice*, that automates most of the process of searching firmware binaries for the presence of logic flaws. To the best of our knowledge, Firmalice is the first firmware analysis system working at the binary level, in a scalable manner, and with no requirement to instrument code on the original device.

We applied Firmalice to the detection of authentication bypass flaws, as seen in several recent, high-profile disclosures of firmware backdoors. To allow Firmalice to reason about such flaws, we created a novel model of authentication bypass vulnerabilities, based around the concept of an attacker's ability to determine the input necessary to execute privileged operations of the device. Intuitively, if an attacker can derive the necessary input for driving a firmware to perform a privileged operation simply by analyzing the firmware, the authentication mechanism is either flawed or bypassable. Additionally, this model allows us to reason about complicated backdoors, such as cases when a backdoor password is secretly disclosed to the user by the firmware itself, as we consider such information determinable by an attacker.

Because the definition of a *privileged operation* (*i.e.,* an operation that requires preliminary authencation) varies between devices, Firmalice requires the specification of a security policy for each firmware sample, to locate such operations. Our system receives a firmware sample and the specification of its security policy, and then loads the firmware sample, translates its binary code into an intermediate representation, and performs a static full-program control and data flow analysis, followed by symbolic execution of firmware slices, to detect the presence of any violations of the security policy.

We evaluated our approach against three real-world firmware samples: a network printer, a smart meter, and a CCTV camera. These devices demonstrate Firmalice's ability to analyze diverse hardware platforms, encompassing both ARM and PPC, among other supported architectures. Additionally, these samples represent both bare-metal binary blobs and user-space programs, and their backdoors are triggered in several different ways.

To summarize, we make the following contributions:

- We develop a model to describe, in an architecture-independent and implementation-independent way, authentication bypass vulnerabilities in firmware binaries. This model is considerably more general than existing techniques, and it is focused around the concept of *input determinism.* The model allows us to reason about, and detect, complicated backdoors, including intentionally-inserted authentication, bugs in authentication code, and missing authentication routines, without being dependent on implementation details of the firmware itself.
- We implement a tool that utilizes advanced program analysis techniques to analyze binary code in complex firmware of diverse hardware platforms, and automate much of the process of identifying occurrences of authentication bypass vulnerabilities. This tool uses novel techniques to improve the scalability of the analysis.
- We evaluate our tool on three real-world firmware samples, detailing our experiments and successfully detecting authentication bypass vulnerabilities, demonstrating that manual analysis is not sufficient for authentication bypass detection.

II. Authentication Bypass Vulnerabilities

The increased prominence of embedded consumer electronics have given rise to a new challenge in access control. Specifically, many embedded devices contain *privileged operations* that should only be accessible by *authorized users.*

[4]The operating system is self-contained in the binary, and we cannot rely on *a-priori* knowledge or known interfaces to such systems.

One example of this is the case of networked CCTV cameras: it is obvious that only an authenticated user should be able to view the recorded video and change recording settings. To protect these privileged operations, these devices generally include some form of user verification. This verification (i.e., only an authorized user can access privileged functionality) almost always takes the form of an authentication of the user's credentials before the privileged functionality is executed.

The verification can be avoided by means of an authentication bypass attack. Authentication bypass vulnerabilities, commonly termed "backdoors," allow an attacker to perform privileged operations in firmware without having knowledge of the valid credentials of an authorized user.

The backdoors that we have analyzed tend to assume one of several forms, which we will detail here, before describing how we designed Firmalice to detect the presence of these vulnerabilities.

Intentionally hardcoded credentials. The most common type of authentication bypass is the presence of hardcoded authentication credentials in the authentication routine of an embedded device. Most commonly, this takes the form of a hardcoded string against which the password is compared (e.g., using `strcmp()`). If the comparison succeeds, access is granted to the attacker. There have been many recent high-profile cases of such backdoors. We discuss one such case, a backdoor in the web interface of a networked CCTV camera [14], in Section IX-B.
In some cases, the credentials might not be directly hardcoded in this manner, but still predictable. One example is a popular model of smart meter, that calculates a "factory access" password by hashing its publicly-known model number [25].

Intentionally hidden authentication interface. Alternatively, an authentication bypass can take the form of a hidden (or undocumented) authentication interface. Such interfaces grant access to privileged operations without the need for an attacker to authenticate. Hidden authentication interfaces have been featured in some recent vulnerabilities [16], [2], and we describe one such case, affecting a popular model of network printer.

Unintended bugs. Sometimes, unintended bugs compromise the integrity of the authentication routine, or allow its bypass completely. One example is command injection: some routers allow unauthenticated users to test connectivity by providing a web interface to the ping binary, and incorrect handling of user input frequently results in command injections.

By analyzing known authentication bypass vulnerabilities in firmware samples, we identified that a lack of *secrecy*, or, inversely, the *determinism* of the input necessary to perform a privileged operation, lies at the core of each one. That is, the authentication bypass exists either because the required input (most importantly, the credentials) was insufficiently secret to begin with (i.e., a comparison with a hardcoded string embedded in the binary), because the secrecy was compromised during communication (for example, by leaking information that could be used to derive a password), or because there was no authentication to begin with (such as the case of an administrative interface, listening, sans authentication, on a secret port).

To reason about these vulnerabilities, we created a model based on the concept of *input determinism*. Our model is a generalization of this class of vulnerability, leveraging the observation that input determinism is a common theme in authentication bypass vulnerabilities. Our authentication bypass model specifies that all paths leading from an entry point into the firmware (e.g., a network connection or a keyboard input handler) to a privileged operation (e.g., a command handler that performs some sensitive action) must validate some input that the attacker cannot derive from the firmware image itself or from prior communication with the device. In other words, we report an authentication bypass vulnerability when an attacker can craft (a possible sequence of) inputs that lead the firmware execution to a privileged operation. Whenever the attacker is able to extract such input from the analysis of the firmware itself, he has found an authentication bypass vulnerability.

This model is considerably more general than existing approaches: it is not important how the actual authentication code is implemented, or, to an extent, what the actual vulnerability is; the analysis needs only to reason about the attacker's ability to determine the input. Note that our model does not require any knowledge of a specific authentication function. In fact, as an interesting special case, our system reports an authentication bypass for all instances where the authentication function is entirely missing.

Unlike classical memory corruption vulnerabilities, such as buffer overflows, logic vulnerabilities such as authentication bypasses require a semantic understanding of the actual device in question. Specifically, the definition of a *privileged operation* will differ for different devices. Firmalice requires the analyst to provide this information as part of a "Security Policy", which specifies resources that a device may not access or actions that a device cannot perform without authentication. We describe these policies in detail in Section V.

In the next section, we will provide an overview of Firmalice's operation, from the input of a firmware sample and security policy to the detection of authentication bypass vulnerabilities.

III. Approach Overview

The identification of authentication bypasses in firmware proceeds in several steps. At a high level, Firmalice loads a firmware image, parses a security policy, and uses static analysis to drive a symbolic execution engine. The results from this symbolic execution are then checked against the security policy to identify violations.

We summarize each individual step in this section, and describe them in detail in the rest of the paper.

Firmware Loading. Before the analysis can be carried out, firmware must be loaded into our analysis engine. We describe this process, and the special challenges that firmware analysis introduces, in Section IV. The output of this step is an internal representation of a loaded, ready-to-analyze firmware sample.

Security Policies. Firmalice has the capability to translate security policies into analyzable properties of the program itself. Specifically, Firmalice takes the *privileged operation*, described by a security policy, and identifies a set of *privileged program points*, which are points in the program that, if executed, represent the privileged operation being performed. Security policies, and how Firmalice translates them into privileged program points, are described in Section V.

Static Program Analysis. The loaded firmware is then passed to the *Static Program Analysis* module. This module generates a program dependency graph of the firmware and uses this graph to create an *authentication slice* from an entry point to the privileged program point. This is detailed in Section VI.

Symbolic Execution. The authentication slice created by the *Static Program Analysis* module is passed to Firmalice's *Symbolic Execution* engine, presented in Section VII. The symbolic execution engine attempts to find paths that successfully reach a *privileged program point*. When such a path is found, the resulting symbolic state (termed the *privileged state*), is passed to the *Authentication Bypass Check* module.

Authentication Bypass Check. Every *privileged state* found by the *Symbolic Execution* engine is passed to the *Bypass Check* module. This module uses the concept of *input determinism* to determine whether the state in question represents the use of an authentication bypass vulnerability. The authentication bypass model, and the procedure to check a privileged state against it, are described in Section VIII. If the state is determined to represent an authentication bypass, Firmalice's analysis terminates, and the input required to trigger the bypass is extracted and provided as Firmalice's output. If the input required to bypass authentication depends on prior communication with the device, Firmalice produces a function that, given the output of such communication, produces a valid input.

State	Constraints	Input
Backdoor	input_0 = “GO” && input_1 = “ON”	“GO\nON\n”
Normal	input_0 = get_username_0 && input_1 = get_password_0	(undetermined)

TABLE I: The privileged states resulting from Firmalice's symbolic execution.

To better explain how Firmalice operates on a firmware sample, we present an example in this section. For simplicity, the example is a user-space firmware sample with a hardcoded backdoor, shown in Listing 1 (the backdoor is the check in lines 2 and 3). Note that while Listing 1 presents source code, our approach operates on binary code.

In this example, the security policy provided to Firmalice is: “The Firmware should not present a prompt for a command (specifically, output the string `Command:`) to an unauthenticated user.”

Firmalice first loads the firmware program, using the techniques described in Section IV, and carries out its Static Program Analysis, as described in Section VI. This results in a control flow graph and a data dependency graph. The latter is then used to identify the location in the program where the string `Command:` is shown to the user. This serves as the privileged program point for Firmalice's analysis. The control flow graph, which is part of the end result of the Static Program Analysis, is shown in Figure 1, with the privileged program point marked with a dashed outline.

Firmalice utilizes its Static Program Analysis module to create an authentication slice to the privileged program point. In our example, this slice comprises the nodes in Figure 1 that are not greyed out.

The extracted authentication slice[5] is then passed to Firmalice's Symbolic Execution engine. This engine explores the slice symbolically, and attempts to find user inputs that would reach the privileged program point. In this case, it finds two such states: one that authenticates the user via the backdoor, and one that authenticates the user properly. The symbolic constraints associated with these states are shown in Table I.

As these privileged states are discovered, they are passed to the Authentication Bypass Check module. In this case, the component would detect that the first state (with a username of “GO” and a password of “ON”) contains a completely deterministic input, and, thus, represents an authentication bypass. Upon detecting this, Firmalice's analysis terminates and outputs the input required to reach the privileged program point.

Listing 1: Example of authentication code containing a hard-coded backdoor.

```
 1 int auth(char *u, char *p) {
 2   if ((strcmp(u, "GO") == 0) &&
 3       (strcmp(p, "ON") == 0))
 4       return SUCCESS;
 5
 6   for (int i = 0; i < 10000000; i++)
 7       pointless();
 8
 9   char *stored_u = get_username();
10   char *stored_p = get_password();
11   if ((strcmp(u, stored_u) == 0) &&
12       (strcmp(p, stored_p) == 0))
13     return SUCCESS;
14   else return FAIL;
15 }
16
17 int main() {
18   puts("Hello!");
19   if (auth(input("User:"), input("Password:")))
20     system(input("Command:"));
21 }
```

IV. Firmware Loading

The first step of analyzing firmware is, of course, loading it into the analysis system. Firmware takes one of two forms:

user-space firmware. Some embedded devices actually run a general-purpose OS, with much of their functionality implemented in user-space programs. A common example of this is the wide array of Wi-Fi routers on the market, generally running a stripped-down version of Linux. All of the OS primitives (i.e., system calls), program entry points, and library import symbols are well-defined.

Binary-blob firmware. Firmware often takes the form of a single binary image that runs directly on the bare metal of the device, without an underlying operating system. OS and library abstractions do not exist in such cases, and it is generally unknown how to properly initialize the runtime environment of the firmware sample, or at what offset to

[5]Starting at the user input in line 19, traversing the `auth()` function, and ending at the privileged program point in line 20.

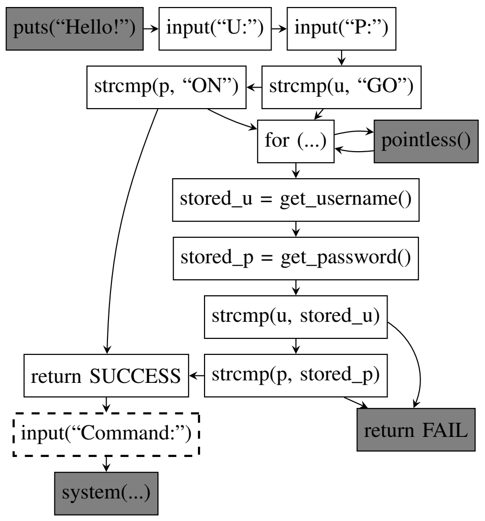

Fig. 1: Firmalice's CFG for the example. The darkened nodes are excluded from the authentication slice.

load the binary and at what address to begin execution.

Analyzing user-space firmware samples is analogous to analyzing a normal user-space program, whereas binary-blob firmware presents several challenges unique to firmware analysis, which we discuss in this section. The output of this phase of the analysis is an internal representation of the firmware, properly loaded in memory and ready to be analyzed. This is then passed to the Static Program Analysis step, described in Section VI.

A. Disassembly and Intermediate Representation

Before an analysis of a firmware sample can be carried out, the binary must be disassembled. This is complicated by the fact that, in the case of binary-blob firmware, the base address where the binary should be loaded and the entry point are not known. Disassembling binary code without this knowledge has been well explored by existing work [19]. Therefore, we leverage existing techniques to acquire a reliable disassembly of the firmware.

As with any static analysis tool, proper disassembly of the firmware sample is a strict requirement for Firmalice's operation. However, modern disassembly techniques have been honed on evasive code, such as malware, and we feel (and, in fact, it has been our experience) that there are no issues disassembling firmware code. Unlike malware, and due to the power and performance requirements of embedded devices, firmware is not obfuscated, and the aforementioned techniques are effective.

Firmalice supports a wide range of processor architectures by carrying out its analyses over an intermediate representation (IR) of binary code. While the choice of a representation itself is not important for our analysis, we present the IR that Firmalice uses in Appendix A.

B. Base Address Determination

Binary-blob firmware typically comes with no information as to the memory location at which it expects to be loaded in the device's memory. Before an analysis of the firmware can be carried out, this value must be determined. Firmalice accomplishes this by leveraging jump tables in the binary.

Jump tables comprise a set of absolute code addresses, stored sequentially in memory. These addresses are read (in many cases, using absolutely-addressed memory accesses) by indirect jumps to determine the jump target. Firmalice identifies the expected location of a binary-blob firmware in memory by analyzing the relationship between jump table positions and the memory access pattern of the indirect jump instructions.

The targets of jump tables tend to exhibit high spatial locality, as they are commonly different cases of a switch statement in the same function. That is, jump tables are typically stored as consecutive values in memory, each of these values being a target address. To identify jump tables, Firmalice scans a binary blob (in steps equal to the architecture's address bit width) for consecutive values that differ only in their least significant two bytes. Firmalice then analyzes all indirect jumps found in the disassembly phase and identifies the memory locations from which they read their jump targets. The binary is then relocated so that the maximum number of these accesses are associated with a jump table.

C. Entry Point Discovery

Unfortunately, without a standard executable file format, binary blobs lack entry point information. That is, even after disassembling a binary, it is unclear from which start instruction the analysis should begin. As an additional complication, there may be multiple entry points to support features such as interrupt requests, with each interrupt request handler representing an additional entry point into the firmware.

Firmalice's static analysis requires knowledge of the entry points to the firmware. Prior work, such as Avatar [33], has required the manual specification of entry points, but in order to reduce the amount of required manual input, Firmalice attempts to automatically identify potential execution entry points. This is done in several steps.

First, Firmalice attempts to identify functions in the binary blob. This is done by scanning through the binary blob for common function prologue instructions (depending on the architecture in question) and analyzing the control flow until a return is encountered. If the function being analyzed issues a call instruction, Firmalice adds the called function to its analysis as well.

Next, Firmalice creates a coarse directed call graph from the list of functions, and identifies all the weakly-connected components of this graph. Any root node of a weakly-connected component is identified as a potential entry point. This is based on the assumption that, since it is not called in the code, it may be called as an interrupt handler. For Firmalice's purposes, an over-estimation of entry points is acceptable in practice. The reason for this is that the privileged program points are not reachable from most of the entry points, and hence the static analysis discards superfluous entry points from further consideration.

V. Security Policies

Traditional vulnerability detection systems such as KLEE [8], AEG [29], and Mayhem [24], among others, are designed to identify memory corruption vulnerabilities in software. Since such vulnerabilities are easily described in a general way (i.e., a control flow hijack occurs whenever the program being analyzed jumps to a user-specified location), these systems can be created with a specific vulnerability model and that is then leveraged in the analysis of many different programs.

Firmalice's task is more difficult, as authentication bypass vulnerabilities are a class of logic flaws. Logic flaws take many forms based on, intuitively, the actual intended logic of the developers of the software (or, in our case, firmware) that is analyzed. Since a logic flaw is a deviation of a program's execution from the logic intended by the developers of the program, what actually constitutes one is highly dependent on what the device in question is designed to do. This holds true for authentication bypass vulnerabilities, the specific class of logic flaws that Firmalice is designed to detect. For example, the ability to watch videos without authentication might be acceptable when dealing with a streaming media set-top box, but represents an authentication bypass when analyzing a network-connected camera.

Automatically reasoning about the *intended* logic of a program requires reasoning about the intentions of the programmer, which we consider outside of the scope of program analysis. Thus, Firmalice requires a human analyst to provide a security policy. For our purposes, a security policy must specify what operations should be considered privileged (and, hence, must always require the user to be authenticated).

When provided a security policy, Firmalice analyzes the firmware in question to convert the policy into a set of *privileged program points*: that is, a set of points in the code of the firmware that, when executed, would cause the privileged operation to be performed. This set of program points is then utilized by Firmalice in its analysis to identify if the execution can reach the specified program point without proper authentication.

These policies vary in the amount of knowledge that they require the analyst to have about the inner working of the firmware: from information that any user moderately familiar with the device would possess, to intricate details about code reachability or memory accesses. The rest of this section describes the policies that Firmalice supports and discusses how Firmalice utilizes these policies to identify *privileged program points*.

Static output. A security policy can be specified as a rule about some static data (usually ASCII text, but in general any sequence of bytes) the program must not output to a user that has not been properly authenticated. An example of such policy is "The program must not output `AUTHENTICATION SUCCEEDED` to an unauthenticated user."
When provided such a policy, Firmalice searches the firmware for the static data and utilizes its data dependency graph (described in Section VI) to identify locations in the program where this data can be passed into an output routine. These locations become the *privileged program points* for the remainder of the analysis.

Behavioral rules. Another policy that Firmalice supports is the regulation of what *actions* a device may take without authentication. In the case of a smart lock, this policy might be "The lock motor must never turn without proper authentication." For Firmalice to be able to reason about such policies, the user must also specify *how* this action would be accomplished. For example, for a device with peripherals that should never read from an attached camera without authentication, this might be "A file in `/dev` must never be opened without authentication."
Firmalice processes this policy by analyzing its control flow graph and data dependency graph for positions where an action is taken that matches the parameters specified in the security policy. In our example, this would be any location where a string that is data-dependent on any string starting with "/dev" is passed to the `open` system call.

Memory access. Embedded devices often communicate with and act on memory-mapped sensors and actuators. To support identifying authentication bypass vulnerabilities in such devices, Firmalice accepts security policies that reason about access to absolute memory addresses. When supplied such a policy, Firmalice identifies locations in the data dependency graph where such memory locations are accessed, and identifies them as *privileged program points*.

Direct privileged program point identification. If the analyst has detailed knowledge about the firmware, the privileged program points can be specified directly as function addressed in the security policy. These are then passed directly to the rest of the analysis.

These security policies are general enough to cover the intended behavior of the firmware samples that we have seen so far.

Of course, Firmalice's Security Policy Parsing module can be extended to support other types of security policies, if required. However, we see the creation and parsing of more intricate security policy as an orthogonal problem to the identification of authentication bypass vulnerabilities, and thus, consider further work in this area outside of the scope of our contribution.

The security policy, along with the firmware sample itself, represent the inputs to Firmalice.

VI. Static Program Analysis

Symbolically executing entire binary firmware images is not feasible due to the size of the firmware of complex embedded devices. Instead of analyzing entire binaries, Firmalice focuses on the portions of binaries that are relevant to authentication bypass vulnerabilities. Specifically, the symbolic execution step only needs to be carried out on the parts of the firmware leading to a *privileged program point* in the firmware. Firmalice isolates this code by creating a *slice* through the firmware. Specifically, Firmalice creates a *backward* slice, starting from the privileged program point, backwards to an entry point in the firmware.

The static analysis module requires as input the loaded firmware sample (produced by the Firmware Loading module, described in Section IV). The actual slicing step also requires the address of one or more *privileged program points*. These should be instructions in the firmware that should only be reached by authenticated users. As we discuss in

Section V, privileged program points are derived from an analyst-provided security policy.

The identification of privileged program points specified by a security policy, and the creation of backward slices leading to them, requires the use of a program dependency graph (PDG) to reason about the control and data flow required to arrive at a specific point in the program. The program dependency graph comprises a data dependency graph (DDG) and a control dependency graph (CDG). Those, in turn, require a control flow graph to be created.

A. Control Flow Graph

The first step in creating a PDG is the creation of a CFG, a graph of program basic blocks and transitions between them. Firmalice creates a context-sensitive CFG by statically analyzing the firmware, starting from each of the entry points and looking for jump edges in the graph. Firmalice can support computed and indirect jumps (including jump tables) by leveraging its *Symbolic Execution* module, described in Section VII. Firmalice's analyses are performed with a *call-site* context sensitivity of 2, to improve the precision of the static analysis. This threshold for the call-site context sensitivity can be changed at the expense of an exponential runtime increase, but, in practice, we have found that a threshold of 2 works well for the firmware samples that we analyzed.

Firmalice leverages several techniques to increase the precision of its control flow graph. During CFG generation, Firmalice utilizes *forced execution* to systematically explore both directions of every conditional branch [32]. When it encounters a computed or indirect jump, Firmalice can leverage its symbolic execute engine (which will be described in Section VII) to reason about the possible targets of that jump. By doing this, Firmalice is able to handle complex control flow transfers, such as jump tables. In turn, a precise CFG has a trickle-down effect on the precision of the rest of Firmalice's analysis.

Firmalice stores the context-sensitive CFG as a graph, in which the *contexts* are nodes and edges represent control flow transfers between these contexts. This means that the graph might contain several distinct instances of a basic block γ with a control transfer edge to basic block α, as long as the call-sites of α and γ differ.

B. Control Dependency Graph

A control dependency graph represents, for each statement X (generally, a binary instruction, but in our case, an IR statement), which other statements Y are guaranteed to execute if X is executed. In turn, this can be used to reason about statements that *must* be executed before a specific given statement is executed.

Again, we use a context sensitivity of 2 when generating the CDG, which allows Firmalice to reason about not only the basic block that needs to execute so that a given statement is reached, but also the call context from which that basic block must be executed. The CDG is generated via a straightforward transformation of the CFG [5].

The CDG is not used directly, but is combined with the data dependency graph to create the PDG.

C. Data Dependency Graph

A data dependency graph (DDG) shows how instructions correlate with each other with respect to the production and consumption of data. Efficiently generating a sound DDG for a binary slice has several challenges. First, program slicing requires a flow-sensitive and context-sensitive data flow analysis, with a runtime complexity exponential to the number of all possible paths in a program. Second, analyzing the data flow of binary programs poses some unique problems. For instance, the precision of the DDG suffers from any imprecision in the CFG from which it is built, and creating a precise CFG statically is a hard problem for arbitrary binary code. Additionally, all information about data structures and types is discarded during compilation, which makes performing a sound data flow analysis even harder. Thus, most data flow analyses are designed to work with high-level languages, but not with binary code. Finally, the analysis result should be sound, otherwise one risks removing instructions that are otherwise required to achieve a proper result.

To handle the issues mentioned above, Firmalice adopts an existing, worklist-based, iterative approach to data flow analysis [30]. The approach is an inter-procedural data flow analysis algorithm that uses *def-use* chains, in addition to *use-def* chains, to optimize the worklist algorithm.

As with the other algorithms in the static analyses phase, the DDG is generated with a context sensitivity of 2.

D. Backward Slicing

Using the program dependency graph, Firmalice can compute backward slices. That is, starting from a given program point, we can produce every statement on which that point depends. This step leverages slicing techniques from existing work in the literature [5]. Slicing is used to improve the feasibility of the symbolic analysis on large binaries, in two ways. First, it removes entire functions that are irrelevant to the analysis. Since symbolic analysis, in the general case, must explore every path of a program, this represents a substantial decrease in analysis complexity. Second, since our IR translates complex instructions into multiple simple statements, Firmalice's slicing allows one to ignore irrelevant side-effects of these instructions. This is especially relevant for architectures that implicitly update conditional flags (specifically, ARM, x86, and AMD64), as it frees Firmalice from the need to evaluate the flag registers when they are not used (which, on such architectures, is the common case).

VII. Symbolic Execution Engine

After an authentication slice is created by the *Static Program Analysis* module, Firmalice attempts to identify user inputs that successfully reach the privileged program point. Recall that an authentication slice is a set of instructions between a proposed entry point and the privileged program point that the attacker tries to reach. To enable our analysis, we have implemented a Symbolic Execution Engine. Our approach to symbolic execution draws on concepts proposed in KLEE [8], FuzzBALL [7], and Mayhem [24], adapted to our specific problem domain.

Specifically, the implementation of this module of Firmalice follows ideas presented in Mayhem, adding support for symbolic summaries of functions (described in paragraph VII-B), to automatically detect common library functions and abstract their effects on the symbolic state. This greatly reduces the number of paths that the symbolic executor must explore, since it prevents such functions from causing the analysis to branch.

We discuss several details specific to our symbolic execution engine in this section.

A. Symbolic State and Constraints

Firmalice's symbolic analysis works at the level of symbolic *states*. A symbolic state is an abstract representation of the values contained in memory (*e.g.*, variables), registers, as well as constraints on these values, for any given point of the program (*i.e.*, each program point has an independent state).

Constraints are expressions limiting the range of possible values for a symbolic variable. They may express relations between symbolic variables and constants (i.e., `x < 5`) or between multiple symbolic values (i.e., `x < y + z`).

For user-space firmware processes, the state also contains other program information, such as the status of open files. States are modified by symbolic translations of IR representations of binary instructions that consume an input state and produce one or, in the case of conditional or computed jumps, multiple output states. As the execution goes following paths in the program, Firmalice keeps tracks of symbolic constraints in a set of *path constraints*. Whenever a path reaches the privileged program point, its associated state is labeled as a *privileged state* and passed to the Authentication Bypass Check module for further analysis, based on constraint solving[6]. The term *constraint solving* refers to the problem of finding concrete or symbolic solutions that satisfy a set of constraints on a variable (e.g., determining, in the case of `x < 5 && x >= 0`, that `x` can be 0, 1, 2, 3, or 4).

B. Symbolic Summaries

Firmalice adopts the concept of "symbolic summaries", a well-known concept in program analysis, which involves descriptions of the transformation that certain commonly-seen functions (or, generally, any piece of code) have on a program state [34]. The intuition behind this concept is that the effects of certain functions can be more efficiently explained through a manual specification of constraints than by analyzing the underlying binary code. This is because an initial analysis of a piece of binary code lacks a *semantic* understanding of what that code is trying to accomplish. A process that had such an understanding, however, could analyze the code as a whole and introduce constraints that took these semantics into account. In fact, we found that such a process has two advantages: properly summarizing the code allows us to avoid branching the analysis state during the execution of such functions, and the constraints that are generated are often simpler than those that would be generated from an analysis of the code itself.

To explore this concept in our analysis, we implemented support for symbolic summaries in Firmalice. A symbolic summary acts in the same way as a binary instruction: it consumes an input state and produces a set of output states. We implemented symbolic summaries for 49 common functions from the Standard C Library.

While this concept is well-known in the field of program analysis, applying it to automatic binary analysis is not trivial, as Firmalice needs to know which pieces of code should be replaced by these summaries. To determine this automatically, we created a set of test cases for each of the functions that we summarized. These test cases, comprising an input state (representing a set of arguments to the function) and a set of checks of its effect on this state, attempt to determine whether or not an arbitrary binary function is an implementation of the function summarized by the symbolic summary in question.

Generally, more than one test case is required to uniquely identify a library function. For example, several different test cases are required to distinguish between `strcmp()` and `strncmp()`, since the two functions act in the same way for certain sets of inputs (lower case strings for example). Similarly, multiple test cases are required to differentiate between `memcpy()` and `strncpy()`. While this represents more work when writing test cases, it also allows us to speed up the testing procedure, because if a function fails a test case that should be passed by both `memcpy()` and `strncpy()`, we can conclude that it is neither of those functions.

When Firmalice symbolically calls a function for the first time (i.e., analyzing a call instruction), the analysis is paused and the function-testing phase begins. Firmalice first attempts to run the function with the test case states. If all of the test cases of a symbolic summary pass, Firmalice replaces the entry point to the function in question with that symbolic summary, and continues its analysis. Any subsequent jumps to that address will instead trigger execution of the symbolic summary. If no symbolic summary is identified as the right summary for a function, the function is analyzed normally. The test cases should be mutually independent across all symbolic summaries. That is, for any given function, if all test cases of symbolic summary A pass, then there must be no summary B for which all test cases also pass. Such situations arise in the case of inadequate test cases, and must be remedied before Firmalice can properly detect symbolic summaries.

While symbolic summaries allow Firmalice to perform a considerably deeper analysis than would otherwise be possible, there is a trade-off. Because we do not fully analyze the summarized code, our approach would miss any backdoors that were hidden in common library functions. We feel that this trade-off is acceptable.

C. Lazy Initialization

Binary-blob firmware contains initialization code that is responsible for setting various memory locations to initial values, setting up request handlers, and performing other housekeeping tasks. However, since Firmalice has no prior knowledge of such code[7] it is not executed before beginning the analysis, leading to complications when, for example, kernel-level functionality of firmware attempts to access certain global data structures. If such data structures are not initialized, superfluous paths based on normally infeasible kernel conditions are introduced into the analysis.

[6]Firmalice utilizes Z3 [13] to perform symbolic constraint solving.

[7]the execution starts after the input related to the authentication routine

To mitigate this, Firmalice adopts a lazy approach to firmware initialization. When the execution engine encounters a memory read from uninitialized memory, it identifies other procedures that contain direct memory writes to that location, and labels them as *initialization procedures*. If an initialization procedure is identified, the state is duplicated: one state continues execution without modification, while the other one runs the initialization procedure before resuming execution. This allows Firmalice to safely execute initialization code without the risk of breaking the analysis.

VIII. Authentication Bypass Check

As discussed in Section III, our model of an authentication bypass builds upon the property of *input determination*. That is, if an attacker can analyze the firmware and produce inputs, possibly including valid authentication credentials, to reach a privileged program point, an authentication bypass is said to exist.

This model is not dependent on the implementation of the backdoor itself, but rather on the fundamental idea behind authentication bypass vulnerabilities: the attacker can create an input that, regardless of the configuration of the device, will allow them to authenticate (*i.e.,* reach a privileged program point).

To detect such bypasses, Firmalice leverages the property of *constraint solvability* with respect to the user input required to achieve authentication. Specifically, we model the *determinism* of the input with the ability to concretize it to a unique value, as described in Section VIII-C. However, we make this determination after taking into account the *exposure* of data from the device, in the form of output to the user. Thus, even in the presence of a challenge-response protocol, Firmalice can detect an authentication bypass vulnerability.

This model can also be expanded to reason about authentication bypasses with a range of valid backdoor credentials. However, as we have not observed this in practice, we did not include such detection in our implementation.

Given an *privileged state* (i.e., the final state of a path that reaches a privileged program point) from the Symbolic Execution engine, the Authentication Bypass Check module identifies the input and output from/to the user and reasons about the *exposure* of data represented by the output. It then attempts to uniquely concretize the user input (*i.e.,* to solve the constraints associated to the user input when the *privileged state* is reached). If the user input can be uniquely concretized, then it represents that the input required to reach the *privileged program point* can be uniquely determined by the attacker, and the associated path is labeled as an *authentication bypass*. At this point, Firmalice terminates its analysis. In cases where the user input depends on data exposed by the device's output, a function that can generate valid inputs for a provided output is produced.

A. Choosing I/O

What should be considered as user input to the firmware (and, similarly, output from the firmware) is not always obvious. For example, devices might have complex interactions with their environment, and receive input in unexpected ways. Therefore, Firmalice uses several heuristics to identify input and output.

If the firmware is a user-space firmware, Firmalice checks for the presence of network connections in the privileged slice. If a connection is found, it is assumed to represent the user input. Alternatively, if no connection is found, user input is assumed to be stdin (file descriptor 0), and output is assumed to be stdout (file descriptor 1).

In the case of a binary blob, Firmalice attempts a concretization on symbolic values coming from every interrupt. If one of these inputs concretizes mainly to ASCII text, it is considered to be the user input. Similarly, any symbolic value passed into an interrupt that concretize mainly into ASCII text, is considered to be the output of the firmware. Alternatively, to avoid these heuristics, Firmalice can accept a specification of the Application Binary Interface (i.e., which interrupts accept output and which provide input) of the firmware and use that to choose between input and output.

B. Data Exposure

The core intuition of our approach is that data seen by the user, via an output routine, is *exposed* to the attacker. While seemingly obvious, this has important implications for authentication bypass detection. Specifically, our intuition is that this exposure does not just reveal information about the output data: information is also revealed about any data that *depends on* or is *related to* the output. For example, if a hash of a user-specified, secret password is revealed to the attacker prior to authentication, it reveals some amount of information about the password itself (in the worst case scenario, such a hash could then be brute-forced and the password would be completely revealed). In essence, we take into account the fact that the attackers can deduce information about authentication credentials by observing program outputs.

We implement this in Firmalice by leveraging its constraint solver and output routine detection. Any data, D, that is passed into an output routine is identified as having been exposed. To model this exposure, we use the constraint solver to retrieve a single concrete solution, C, for D, and add the constraint $D == C$ to the constraint set. Adding this constraint has an effect on the concrete solutions associated with other symbolic variables (for example, if a symbolic variable E previously existed with a constraint $E == D$, then the constraint $D == C$ also implies $E == C$). This represents any loss of secrecy that these variables experience from the revelation of D to the attacker.

To avoid false positives from after-the-fact credential revelation on the part of the firmware, Firmalice only applies this policy to data that is output *before* any user input is received.

C. Constraint Solving

For each privileged state, Firmalice attempts to concretize the user input to determine the possible values that a user can input to successfully reach the privileged program point. A properly-authenticated path contains inputs that concretize to a large set of values (because the underlying passwords that they are compared against are unknown, and thus, unconstrained). Conversely, the existence of a path for which the input concretizes into a limited set of values (for simplicity, and from investigating existing examples of backdoors, we set this threshold to 1) signifies that an attacker can determine, using a combination of information within the firmware image and information that is revealed to them via device output, an input that allows them to authenticate.

Since Firmalice limits its analysis to the authentication slice itself, irrelevant data is not included in the produced user input. This makes Firmalice resilient to cases that would be *arbitrarily non-deterministic*, such as when some data from the user is ignored or not used (and, thus, concretizes to no specific value). While this means that Firmalice's output might not be directly re-playable to achieve authentication bypass, this functionality is outside of the scope of our design.

IX. Evaluation

We evaluated Firmalice by vetting three devices for authentication bypass vulnerabilities, two of which had actual backdoors. These devices, the Schneider ION 8600 smart meter, the 3S Vision N5072 CCTV camera, and the Dell 1130n Laser Mono Printer, represent a wide range of devices of disparate architectures. ARM (both little-endian and big-endian) and PPC are both represented, as are both binary-blob and user-space program firmware styles. Additionally, the devices have widely different authentication processes.

We chose these devices because the authentication vulnerabilities that they contain were already discovered manually, and, since these vulnerabilities have already been released, we are not endangering the users by discussing them (and providing examples). We chose three devices because, despite the fact that Firmalice's analysis is automated, a security policy needs to be provided for each device. This represents some manual work, and a truly large-scale study was infeasible. Additionally, collecting and unpacking firmware samples is extremely complicated to automate. Firmware is shipped in many different, non-standard formats, and the process to download firmware images is frequently complicated, and varies from vendor to vendor. While this is an addressable problem, as shown by Costin *et al.* [10], we consider it outside of the scope of our work. However, we feel that these samples represent Firmalice's applicability to different devices of different architectures.

In this section, we will describe each firmware, then detail their user interaction, present our analysis results, and describe any backdoors that Firmalice identified. Aside from the device-specific uses of these backdoors, each one can also be used as a pivot point into the victim's network. The nature of some of these devices means that they are frequently either physically positioned *outdoors*, exposed directly to the Internet, or are otherwise not closely monitored, making them a prime target for attackers.

We carried out this evaluation on our prototype of Firmalice, comprising over 14,000 lines of Python and 3,000 lines of C. Our implementation is single-threaded, although the approach itself would scale near-linearly in the symbolic analysis phase. Thus, the execution time presented in this section is representative of what can be accomplished using a *single node* of Firmalice, and significant improvements in runtime can be achieved by parallelizing the symbolic execution.

Measurement	ION	3S	Dell
Total size (KB)	1,988	1,264	7,172
Basic blocks (total)	74,808	10,354	151,005
Basic blocks (slice)	1,144	212	532
Slice (statements)	56,977	7,808	24,387
Static analysis time (seconds)	2,323	315	857
Symbolic execution time (minutes)	12	26	705

TABLE II: The results of Firmalice's analysis for the ION 8600, the 3S Vision N5072 and the Dell 1130n.

A. Schneider ION 8600 Smart meter

As the smart meter market exploded worldwide, Schneider Electric corporation released the ION 8600, a smart meter model meant for both residential and commercial use. Such devices play a privacy-critical and safety-critical role: the information that they process can be used to determine the habits of a home's resident, and any malicious tampering can cause extremely dangerous situations due to the amount of electricity involved.

A researcher from IOActive Labs presented a backdoor in the Schneider ION 8600 smart meter model at BlackHat in 2012 [25]. The backdoor was identified through manual static analysis of the firmware. Schneider Electric acknowledged the backdoor in a press release [3] and released an updated firmware image. Our interpretation of the presentation by IOActive, and the press release by Schneider, led us to think that the backdoor was remotely exploitable.

We saw this as a great opportunity to verify Firmalice's functionality. Even better, the described authentication procedure is relatively complex: rather than being a simple comparison against a hardcoded string, it relies on the exposure of the backdoor credentials (which are dynamically generated by hashing the serial number of the device) to the user during the authentication process. Detecting this type of authentication bypass requires reasoning about the *determinism* of the authentication credentials, in relation to information provided by the device during the authentication process.

The security policy. We observed that the ION would output the string "Access Granted" upon a successful authentication by a user. This was leveraged for the security policy: we set the authenticated point to the location in the firmware where "Access Granted" was printed.

The analysis. This firmware's binary blob contained 1,988 kilobytes of binary code spanning 74,808 basic blocks. The static analysis completed in about 38 minutes, and the resulting authentication slice contained 1,144 basic blocks and 56,977 statements.

The authentication slice identified by Firmalice ran from the input routine to the privileged point. Because the ION's firmware places a bound on the size of the user input, and because symbolic summaries of functions greatly reduce the number of branches that Firmalice must analyze, Firmalice was able to *exhaustively* analyze all paths through the authentication slice. This symbolic analysis ran in 12 minutes, analyzing 1,029,156 statements in 23,044 blocks across all analyzed paths. We present these results in Table II.

To our surprise, Firmalice's analysis yielded no bypasses.

Since symbolic analysis is, in practice, not sound, we assumed that our system must have missed the vulnerability. We manually analyzed the firmware sample, and even attempted the bypass on an acquired device with the vulnerable firmware (verified by the build number and release date) to try to figure out where Firmalice was getting confused. It turned out that, due to complex logic in the authentication routine (which spanned several nested functions with intricate interactions), a user had to be already authenticated with *valid* credentials before using the hardcoded credentials identified by IOActive. Using the hardcoded credentials, after an actual, secure authentication, would grant access to more features of a device. However, the backdoor account could not be accessed from the Internet, unless the attacker already had the user's actual, valid credentials. Therefore, there was no remotely accessible backdoor.

We contacted the IOActive researcher, and he confirmed that we were mistaken in our interpretation. We feel that this anecdote demonstrates the need for a more automated solution: even with manual analysis, it took us a significant amount of time to verify the results of our analysis due to the complexity of the code involved. Given the difficulty (and cost) involved in updating firmware on embedded devices, such mistakes can represent a real financial impact, and a system to automate parts of this analysis can be extremely valuable.

B. 3S Vision N5072 Camera

The 3S Vision N5072 is a CCTV camera with networking functionality.

In April 2014, Craig Heffner presented backdoors in several common embedded devices at the EELive 2014 conference [11]. Among them was the N5072 camera from 3S Vision. This backdoor, which takes the form of a hardcoded authentication credential, allows an attacker to control and view the camera over the network. Especially given the zooming capability of this camera, such an attack can have serious implications with regards to privacy intrusion.

The camera is built on a little-endian ARM architecture. We found that the firmware of this camera is actually an embedded Linux system, comprising Busybox and several camera-specific binaries, including a custom web server.

The security policy. Our security policy for this firmware reflected the purpose of the device itself: the user must not be able to view camera footage without authentication. However, the footage itself was not static, so we could not directly use it for the policy. Instead, we used the static string "Image Type:", which was included when requesting footage from the camera's web interface.

The analysis. Firmalice was able to identify the backdoor in the `httpd` binary, in a total of 31 minutes. This binary, and the libraries that it depends on, contain a total of 1,264 kilobytes of binary code spanning 10,354 basic blocks. The static analysis completed in 315 seconds, and the resulting authentication slice contained 3,553 statements from a total of 7,808 in the corresponding 212 basic blocks. The detection of the backdoor took just over 26 minutes, analyzing 550,660 statements in 34,544 blocks across all executed paths. We present these results in Table II.

The Backdoor. The backdoor in the N5072 was a hardcoded authentication credential during HTTP authentication. The backdoor allows an attacker to stream video from the camera and modify the camera's configuration. Firmalice provided an HTTP request that would be sufficient to reach the privileged program point, in which an "authorization" parameter is passed in the query string. The base64 decoding of the authorization query string parameter is "3sadmin:27988303", which is the hardcoded username and password of the backdoor. Interestingly, Firmalice also stumbled upon a benign bug in the URL parsing code of the camera: query string parameters are parsed, even without the presence of the "?" character that denotes the start of a query string, if the provided query path is blank.

C. Dell 1130n Printer

The Dell 1130n is a network-connected laser printer popular in many office and academic settings. Such printers are frequently connected directly to the Internet, with no protection or filtering in place. In fact, in January 2013, researchers made headlines by pointing out the presence of 86,800 network printers that could be found in a Google search [1].

A backdoor affecting a range of printers manufactured by Samsung, including the Dell 1130n, was discovered in 2012 [2]. This backdoor allows an attacker to change printer settings, intercept documents sent to the printer, and use the printer as a pivot point into the victim's network. The backdoor is triggered by sending a specially-crafted SNMPv1 packet to the printer, with a hardcoded *community string*. This attack works even when SNMP is turned off.

This printer runs on a big-endian ARM CPU, and its firmware is a modified VxWorks binary-blob containing 7,172 kilobytes of binary code across 151,005 basic blocks.

The security policy. We used the printer to evaluate our more fine-grained security policy, defining a memory region, containing configuration parameters, that should not be changed by unauthenticated users. Firmalice identified all program points that write to this memory region, and tagged them as *privileged program points* for the analysis.

The analysis. Firmalice finished its static analysis in just over 14 minutes, and created an authentication slice that contained 13,592 of the total 24,387 IR statements in 532 blocks. The analysis of the slice took 11 hours and 45 minutes, executing a total of 134,536,875 statements in 4,264,568 blocks across all of the analyzed paths. The results are presented Table II.

The Backdoor. The backdoor in the 1130n took the form of a specially crafted SNMP packet, allowing the attacker administrative access to the printer. Firmalice provided an input representing the SNMP packet that would let the attacker reach the privileged program point.

X. Discussion

In this section, we discuss the implications and the limitations of Firmalice and muse about several ideas for future research directions.

Firmalice's target application is the analysis of authentication bypass vulnerabilities in firmware. In general, such software is not actively evasive (unlike, for example, traditional malware), and lends itself well to static analysis.

However, it is possible that a malicious firmware author could attempt to attack Firmalice's analysis. There are two main attack surfaces: the static program slicing and the symbolic execution. Obfuscated firmware could frustrate the former, while specially-crafted operations (designed to overwhelm the constraint solver) could attack the latter. These are weaknesses inherent to any tool based on static slicing or symbolic execution, and Firmalice is also vulnerable to them. Given the status quo in firmware, the presence of such obfuscation or evasive code would be, by itself, an excellent indicator of maliciousness, which Firmalice could be adjusted to detect.

As an alternative to binary obfuscation, malicious firmware authors could attempt to evade Firmalice's input determinism by performing *irreversible* operations. For example, Firmalice's constraint solving module would be unable to solve the constraints generated by a secure hash function, as doing that would be equivalent to reversing the function. As a result, Firmalice, in its current implementation, can be evaded by an authentication bypass that compares a hash of the user's password against the hash of a hard-coded password. A possible mitigation of this evasion is the replacement of the hash function with a symbolic summary that performs a reversible "summary hash". In the case of SHA-256, such a summary hash might simply expand or truncate the input to 256 bits. With this summary hash replacing the original hash function, the constraints generated by Firmalice would be reversible, and the required user input could be identified. However, this represents a large sacrifice in accuracy of the analysis, and false positives could be introduced as a result.

There are also other types of backdoors that Firmalice might fail to detect. Specifically, math-based backdoors with multiple solutions (e.g., "the password must be an integer that is divisible by 10") would, as a result of having multiple valid solutions, be considered as a "correct" authentication. To reason about such backdoors, Firmalice would need to reason about how *restricted* a set of solutions is. This ability would involve extra complexity related to constraint solving and we feel that this analysis is outside of the scope of this research.

Throughout Firmalice's design, we had to make many trade-offs between soundness and scalability. Symbolic analysis, in general, is infeasible to perform with full soundness, because doing this would mean, in the general case, following every path through a program. This would be exponential in the number of branches, and Firmalice makes trade-offs, similar to other tools in the field.

Many of the challenges that Firmalice must deal with could be addressed through the use of dynamic execution monitoring. For example, Firmalice's entry point detection would be unnecessary if the entry point could be deduced from observing the boot process of the device. However, the difficulty of this ranges from extremely complex to impossible for most devices. Since many embedded devices require their firmware to be signed by the device manufacturer, loading custom analysis code (such as that required by Avatar [33]) would require bypassing this protection. Even if this limitation could be bypassed, the disparity between different devices would necessitate a significant implementation effort to analyze each new device, limiting the possible scale of such a system's analysis.

While Firmalice is geared towards detecting authentication bypass vulnerabilities in firmware, the core approach lends itself to any logic flaws that can be similarly modeled. One potential direction of research is a formal language to enable the specification of custom logic flaws for Firmalice to locate. In fact, the Defense Advanced Research Projects Agency has launched a project to explore exactly this, with the goal of assuring the security of embedded devices [4]. DARPA's goal is to eventually be able to specify such models as Natural Language statements that can be converted into logic flaw descriptions.

Firmalice, and symbolic analysis in general, can be greatly improved by a better approach to symbolic loop analysis. When analyzed symbolically, a loop has the potential to branch analysis states at each iteration (one that exits the loop and one that does not), causing a state explosion. Firmalice partially mitigates this through the use of its symbolic summaries, as many of the loops encountered during a program's execution are actually within common library functions. However, in the general case, advances in loop analysis would directly benefit Firmalice's (and other analyzers') analyses.

XI. Related Work

While a number of previous efforts have been focusing on analyzing binary applications on commodity software and hardware platforms, including general frameworks such as Valgrind [23], BitBlaze [28], and Pin[21], as well as symbolic execution based frameworks like AEG [29] and Mayhem [24], focusing on automatic exploit generation on binary programs, the case of embedded firmware received little attention and remains challenging. Among existing research on firmware analysis, the current systems either require access to the source code [12] (which in the case of embedded systems is rarely available), or to the physical device [26], [33].

Schuster *et al.* [26], proposed an approach for automatically identifying software backdoors in binary applications running on x86, x64, and MIPS32 architectures. This approach targets flawed authentication routines as well as commands and services hidden in server-side binaries such as FTP and SSH. The approach builds on top of execution monitoring using GDB, and requires actual execution of the binaries on the target physical system, making it difficult to generally apply the technique to embedded devices. While this work proposes a practical approach to detecting backdoors, it is limited to a specific kind of authentication bypass technique where pointers to *handlers* are actually present as-is in memory[8]. Additionally, Schuster models authentication bypass as a control flow problem, leaving them unable to reason about authentication bypasses resulting from disclosed credentials or buggy authentication routines. Not only is our system able to analyze binaries with no hardware requirements, but our symbolic execution approach also targets a wider range of malicious behaviors. In fact, the *authentication deciders* and *command handlers* that Schuster's approach identifies during the analysis can be used as a security policy by Firmalice, allowing Firmalice to vet the firmware against complex authentication bypass vulnerabilities.

Avatar [33] is a framework supporting dynamic analysis of firmware in embedded systems. It is a hybrid approach,

[8]This approach does not address the cases of obfuscated or indirect addresses to such *handlers*.

involving both the target physical device as well as an emulator based on the selective symbolic execution engine S2E [9]. Communication between the emulator and the target is orchestrated in such a way that I/O operations can be forwarded and executed on the actual hardware and interrupts injected into the emulator. Arbitrary context switches are also supported: execution can be started on the real device and transfered to the emulator for analysis from a specific point in the firmware. Returning execution to the hardware is also supported. In both cases, the execution state is frozen and transferred from/to the hardware or the emulator. While Avatar presents promising capabilities and support for reverse engineering and vulnerability discovery, it requires access to the physical hardware, either through a debugging interface, or by installing a custom proxy in the target environment, which is generally not possible, *e.g.,* in the presence of locked hardware. Our framework is an alternative to such hardware-dependent approaches, by providing a model along with tools for analyzing such firmware with no hardware requirements.

FIE [12] is a platform for detecting bugs in firmware running on the MSP430 family of micro-controllers, mainly focusing on memory safety issues. The source code of the analyzed programs is compiled into LLVM bytecode, which is then analyzed using a symbolic execution engine based on KLEE [8]. The latter has been modified to support the target 16-bit architecture, its memory specification, and its interrupt library. FIE supports hardware specific layouts of memory and access to hardware through *special memory*. It also considers the execution of enabled interrupts at any given point in the program. It performs *complete* analysis of firmware images (*i.e.,* all possible execution paths are taken). In order to achieve this without falling into infinite loops or state explosion, *state pruning* is used, removing redundant (equivalent) states from the list of states to explore, and *memory smudging* is used to concretize variables with respect to a given finite set of values. FIE is limited to analyzing small firmware written in C, for which the source code is available. In comparison, our current work is not bound to any specific architecture (in fact, our symbolic execution engine currently supports multiple architectures) and works directly on binary code with no source code requirement.

Recent advancements have also been made in the field of automated firmware analysis. Costin *et al.* [10] carried out an analysis of over 30,000 firmware samples. However, their system performs no in-depth analysis: it instead extracts each firmware sample and investigates it for artifacts such as included private encryption keys and "known-bad" strings (i.e., known values of hardcoded authentication credentials). This latter action makes the system quite well-suited for discovering backdoors in devices whose firmware shares a codebase with devices that have known backdoors, but not for in-depth analysis of individual firmware samples. With a further investment into analysis automation, Costin's system could be used as an input to Firmalice, allowing for large-scale, automated, in-depth firmware analysis.

XII. Conclusion

We presented Firmalice, a framework for detecting authentication bypass vulnerabilities in binary firmware, for which no source code, and possibly no access to the underlying hardware, is available. Additionally, we have presented a model of authentication bypass vulnerabilities (or backdoors), based on the concept of *input determinism* and have shown that Firmalice is capable of successfully detecting such vulnerabilities in the firmware of two commercially-available systems. Finally, we have demonstrated that current techniques for identifying authentication bypass in firmware, which are mostly limited to manual analysis, are error-prone and insufficient.

Acknowledgements. This work is sponsored by DARPA under agreement number N66001-13-2-4039 and by the ARO through MURI grant W911NF-09-1-0553. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Additionally, we would like to thank Timon Van Overveldt for his assistance with the selection, extraction, and initial analysis of firmware samples.

References

[1] 86,800 network printers open to the whole Internet. http://nakedsecurity.sophos.com/2013/01/29/86800-printers-open-to-internet/.

[2] CVE-2012-4964. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4964.

[3] Re: ION Meter Security. http://www.powerlogic.com/literature/IONMeterCyberSecurityApril2012.pdf.

[4] Vetting Commodity IT Software and Firmware (VET). http://www.darpa.mil/Our_Work/I2O/Programs/Vetting_Commodity_IT_Software_and_Firmware_(VET).aspx.

[5] Akos Kiss, Judit Jasz, Gabor Lehotai, and Tibor Gyimothy. Interprocedural static slicing of binary executables. In *Source Code Analysis and Manipulation*, pages 118–127. IEEE, 2003.

[6] Arstechnica. Bizarre Attack Infects Linksys Routers With Self-Replicating Malware, 2014. http://arstechnica.com/security/2014/02/bizarre-attack-infects-linksys-routers-with-self-replicating-malware/.

[7] Babić, Domagoj and Martignoni, Lorenzo and McCamant, Stephen and Song, Dawn. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22. ACM, 2011.

[8] Cadar, Cristian and Dunbar, Daniel and Engler, Dawson R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI*, volume 8, pages 209–224, 2008.

[9] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 47(4):265–278, Mar. 2011.

[10] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis. A large-scale analysis of the security of embedded firmwares.

[11] Craig Heffner. Finding and Reversing Backdoors in Consumer Firmware. http://www.devttys0.com/wp-content/uploads/2014/04/FindingAndReversingBackdoors.pdf.

[12] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the USENIX Security Symposium*, pages 463–478. USENIX Association, 2013.

[13] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] /dev/ttypS0. Finding and Reverse Engineering Backdoors in Consumer Firmware, 2014. http://www.devttys0.com/wp-content/uploads/2014/04/FindingAndReversingBackdoors.pdf.

[15] /dev/ttyS0. From China, With Love, 2013. http://www.devttys0.com/2013/10/from-china-with-love/.

[16] /dev/ttyS0. Reverse Engineering a D-Link Backdoor, 2013. http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/.

[17] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.

[18] Forbes. When "Smart Homes" Get Hacked: I Haunted A Complete Stranger's House Via The Internet, 2013. http://www.forbes.com/sites/kashmirhill/2013/07/26/smart-homes-hack/.

[19] C. Kruegel, W. K. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the USENIX Security Symposium*, 2004.

[20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (PLDI), 2005.

[22] P. Mag. U.S. Barely Cracks List of Countries With Top Wi-Fi Penetration, 2012. http://www.pcmag.com/article2/0,2817,2402672,00.asp.

[23] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[24] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[25] R. Santamarta. Here be backdoors: A journey into the secrets of industrial firmware. In *BlackHat*, 2012.

[26] F. Schuster and T. Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, (CCS), pages 851–862, New York, NY, USA, 2013. ACM.

[27] Y. Shoshitaishvili. PyVEX - Python bindings for VEX IR. http://github.com/zardus/pyvex.

[28] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information Systems Security*, ICISS, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the network and Distributed System Security Symposium*, Feb. 2011.

[30] Tok, Teck Bok and Guyer, Samuel Z and Lin, Calvin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Compiler Construction*, pages 17–31. Springer, 2006.

[31] Tripwire. SOHO Wireless Router (In)security, 2014. http://www.tripwire.com/register/soho-wireless-router-insecurity/.

[32] L. Xu, F. Sun, and Z. Su. Constructing precise control flow graphs from binaries, 2010.

[33] Zaddach, Jonas and Bruno, Luca and Francillon, Aurelien and Balzarotti, Davide. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the Network and Distributed System Security Symposium*, 2014. http://www.eurecom.fr/publication/4158.

[34] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, 2011.

APPENDIX A
IR TRANSLATION

Because firmware is made for devices with widely diverse architectures, firmware analysis systems must be able to carry out their analysis in the context of many different hardware platforms. To address this challenge, Firmalice translates the machine code of different architectures into an intermediate representation, or *IR*. The IR must abstract away several architecture differences when dealing with different architectures:

Register names. The quantity and names of registers differ between architectures, but modern CPU designs hold to a common theme: each CPU contains several general purpose registers, a register to hold the stack pointer, a set of registers to store condition flags, and so forth. The IR must provide a consistent, abstracted interface to registers on different platforms.

Memory access. Different architectures access memory in different ways. For example, ARM can access memory in both little-endian and big-endian modes. The IR must be able to abstract away these differences.

Memory segmentation. Some architectures, such as x86, which is beginning to be used in embedded applications, support memory segmentation through the use of special segment registers. The chosen IR needs to be able to model such memory access mechanisms.

Instruction side-effects. Most instructions have side-effects. For example, most operations in Thumb mode on ARM update the condition flags, and stack push/pop instructions update the stack pointer. Tracking these side-effects in an *ad hoc* manner in the analysis would be error-prone, so the IR should make these effects explicit.

There are many existing intermediate representations available for use, including REIL [17], LLVM IR [20], and VEX, the IR of the Valgrind project [23]. We decided to utilize VEX due to its ability to address our IR requirements and an active and helpful developer community. However, our approach would work with any intermediate representation. To reason about VEX IR in Python, we implemented Python bindings for libVEX. We have open-sourced these bindings [27] in the hope that they will be useful for the community.

VEX is an architecture-agnostic representation of a number of target machine languages, of which the x86, AMD64, PPC, PPC64, MIPS, MIPS64, ARM (in both ARM and Thumb mode), ARM64, and S390X architectures are supported. VEX abstracts machine code into a representation designed to make program analysis easier by modeling instructions in a unified way, with explicit modeling of all instruction side-effects. This representation has four main classes of objects.

Expressions. IR Expressions represent a calculated or constant value. This includes values of memory loads, register reads, and results of arithmetic operations.

Operations. IR Operations describe a *modification* of IR Expressions. This includes integer arithmetic, floating-point arithmetic, bit operations, and so forth. An IR Operation applied to IR Expressions yields an IR Expression as a result.

Temporary variables. VEX uses "temporary variables" as internal registers: IR Expressions are stored in temporary variables between use. The content of a temporary variable can be retrieved using an IR Expression.

Statements. IR Statements model changes in the state of the target machine, such as the effect of memory stores and register writes. IR Statements use IR Expressions for values they may need. For example, a memory store statement uses an IR Expression for the target address of the write, and another IR Expression for the content.

Blocks. An IR Block is a collection of IR Statements, representing an extended basic block in the target architecture. A block can have several exits. For conditional exits from

IR Expression	Evaluated Value
Constant	A constant value.
Read Temp	The value stored in a VEX temporary variable.
Get Register	The value stored in a register.
Load Memory	The value stored at a memory address, with the address specified by another IR Expression.
Operation	A result of a specified IR Operation, applied to specified IR Expression arguments.
If-Then-Else	If a given IR Expression evaluates to 0, return one IR Expression. Otherwise, return another.
Helper Function	VEX uses C helper functions for certain operations, such as computing the conditional flags registers of certain architectures. These functions return IR Expressions.

TABLE III: A list of relevant VEX IR Expressions for Firmalice's analysis.

IR Statement	Effect
Write Temp	Set a VEX temporary variable to the value of the given IR Expression.
Put Register	Update a register with the value of the given IR Expression.
Store Memory	Update a location in memory, given as an IR Expression, with a value, also given as an IR Expression.
Exit	A conditional exit from a basic block, with the jump target specified by an IR Expression. The condition is specified by an IR Expression.

TABLE IV: A list of relevant VEX IR Statements for Firmalice's analysis and their effects on the firmware state.

ARM Assembly	VEX Representation

subs R2, R2, #8	t0 = GET:I32(16)
	t1 = 0x8:I32
	t3 = Sub32(t0,t1)
	PUT(16) = t3
	PUT(68) = 0x59FC8:I32

TABLE V: An example of a VEX IR translation of a machine code instruction located at 0x59FC4. VEX converts register names to numerical identifiers: 16 refers to R2 and 68 refers to the program counter.

the middle of a basic block, a special "Exit" IR Statement is used. An IR Expression is used to represent the target of the unconditional exit at the end of the block.

Relevant IR Expressions and IR Statements for an analysis are detailed in Tables III and IV.

The IR translation of an example ARM instruction is presented in Table V. In the example, the subtraction operation is translated into a single IR block comprising 5 IR Statements, each of which contains at least one IR Expression. Register names are translated into numerical indices given to the *GET* Expression and *PUT* Statement. The astute reader will observe that the actual subtraction is modeled by the first 4 IR Statements of the block, and the incrementing of the program counter to point to the next instruction (which, in this case, is located at 0x59FC8) is modeled by the last statement.