Backdoors: Definition, Deniability and Detection

Sam L. Thomas, Aurélien Francillon

Backdoors: Definition, Deniability and Detection

Sam L. Thomas[1,2] and Aurélien Francillon[3]

[1] Univ Rennes, CNRS, IRISA, France
[2] University of Birmingham, United Kingdom
m+research@kali.ai
[3] EURECOM, France
aurelien.francillon@eurecom.fr

Abstract. Detecting backdoors is a difficult task; automating that detection process is equally challenging. Evidence for these claims lie in both the lack of automated tooling, and the fact that the vast majority of real-world backdoors are still detected by labourious manual analysis. The term backdoor, casually used in both the literature and the media, does not have a concrete or rigorous definition. In this work we provide such a definition. Further, we present a framework for reasoning about backdoors through four key components, which allows them to be modelled succinctly and provides a means of rigorously defining the process of their detection. Moreover, we introduce the notion of deniability in regard to backdoor implementations which permits reasoning about the attribution and accountability of backdoor implementers. We show our framework is able to model eleven, diverse, real-world backdoors, and one, more complex backdoor from the literature, and, in doing so, provides a means to reason about how they can be detected and their deniability. Further, we demonstrate how our framework can be used to decompose backdoor detection methodologies, which serves as a basis for developing future backdoor detection tools, and shows how current state-of-the-art approaches consider neither a sound nor complete model.

Keywords: Backdoors · Formalisation of definitions · Program analysis

1 Introduction

The potential presence of backdoors is a major problem in deploying software and hardware from third-parties. Recent studies and research has shown that not only powerful adversaries [3], but consumer device manufacturers [2,5] have inserted deliberate flaws in systems that act as backdoors for attackers with knowledge of those flaws. Unlike the exploitation of traditional vulnerabilities whereby a *weird*, unintended program state is reached, backdoors also manifest as explicit, intentional, essentially *normal* program functionality – making their detection significantly more challenging.

Many backdoors are considered by their manufacturers to be accidental, left-over "debug" functionality, or ways to implement software configuration updates without explicit user authorisation [5]. In other cases, device manufacturers deploy firmware coupled with third-party software that introduces backdoor functionality into their otherwise backdoor free systems without their knowledge [9].

The term "backdoor" is generally understood as something that intentionally compromises a platform, aside from this, however, there has been little effort

This article is based upon work supported by COST Action IC1403 (CRYPTACUS).

to give a definition that is more rigorous. To give such a definition is difficult as backdoors can take many forms, and can compromise a platform by almost any means; e.g., a hardware component, a dedicated program or a malicious program fragment. This lack of a rigorous definition prohibits reasoning about backdoors in a generalised way that is a premise to developing methods to detect them. Further hampering that reasoning – especially in the case of backdoors of a more complex, or esoteric nature – is the sheer lack of real-world samples. Documented real-world backdoors are generally simplistic, where their trigger conditions rely upon a user inputting certain static data, e.g., hard-coded credentials. Such backdoors have been studied in the literature with various tools providing solutions relying on varying degrees of user interaction [19,21].

2 Overview

This work provides first and foremost a much needed rigorous definition of the term backdoor: which we view as an intentional construct inserted into a system, known to the system's implementer, unknown to its end-user, that serves to compromise its perceived security. We propose a framework to decompose and componentise the abstract notion of such a backdoor, which serves as a means to both identify backdoor-like constructs, and reason about their detection. While the primary focus of this work is software-based backdoors, by modelling a backdoor abstractly, our framework is able to handle all types of backdoor-like constructs, irrespective of their implementation target.

Many backdoors found in the real-world fall into a grey area as to whom is accountable for their presence; to address this, we define the notion of deniability. We model deniability by considering different views of a system: that of the implementer, the actual system, and the end-user; this allows us to – depending on where backdoor-like functionality has been introduced – reason about if that functionality is a deniable backdoor, accidental vulnerability, or intentional backdoor. In many cases, attempting to model this intention, or the lack thereof, is something that is social or political, thus we do not address such cases in this work, instead we focus on the technical aspects of a backdoor-like functionality.

We show that under our definitions, many backdoors publicly identified are not deniable and thus, their manufacturers should be held accountable for their presence. Aside from manual analysis, little work has been performed to address the detection of backdoors. We perform a study of both academic and real-world backdoors and consider existing methods that can be used to locate backdoor components, as well as how those methods can be improved.

2.1 Contributions

To summarise, the contributions of this work are as follows:

1. We provide a rigorous definition for the term backdoor and the process of backdoor detection.
2. We provide a framework for decomposition of backdoor-like functionality, which serves as a basis for their identification, and reasoning about their detection.

3. We express the notion of deniable backdoors by considering different views of a system: the developer's perspective, the actual system, the end-user, and a user analysing the system.
4. We show examples of both academic and real-world backdoors expressed in terms of our definitions and reason about their deniability and detectability.
5. We demonstrate how our framework can be used to reason about backdoor detection methodologies, which we use to show that current state-of-the-art tools do not consider a complete model of what a backdoor is, and as a result, we are able identify limitations in their respective approaches.

2.2 Related Work

Coverage of complex backdoors is scarce in the academic literature. Tan et al. [20] encode backdoor code fragments using specially-crafted interrupt handlers, which, when triggered, manipulate run-time state, and when chained together, can perform arbitrary computations in a stealthy manner. Andriesse et al. [14] use a cleverly disguised memory corruption bug to act as a backdoor trigger and embed misaligned code sequences into the target executable to act as a payload. Zaddach et al. [24] describe the design and implementation of a hard-drive firmware backdoor, which enables surreptitious recovery of data written to the disk. More complex backdoors have been documented outside of the literature, e.g., those classified as"NOBUS" (i.e., NObody But US) vulnerabilites by the NSA [7], and those associated with APT actors (e.g., [8]).

A related area, that of so-called *weird machines*, describes how an alternative programming model that facilitates latent computation can arise within a program, or system. Both [16] and [18] present such models, as well as how *normal* systems can be forced to execute programs written in those models. In both cases, those models provide a means to implement backdoor-like functionality. Dullien [15] addresses the problem of formalising the term *weird machine*, the relationship between exploitation and *weird machines*, and introduces the concept of provable exploitability. He argues that it is possible to model a program, or system using a so-called Intended Finite-State Machine (IFSM), and in doing so, view a piece of software as an emulator for a specific IFSM. Further, he demonstrates that it is possible to create security games to reason about the security properties of a system by reasoning about it at the level of the states and transitions of its IFSM. His model serves as inspiration for this work.

Zhang et al. [25] explore the notion of backdoor detection and give a first informal definition of the term backdoor. They define a backdoor as "a mechanism surreptitiously introduced into a computer system to facilitate unauthorised access to the system", which while largely agreeing with the current usage of the term, is very high-level and says nothing about the composition of such constructs. Wysopal et al. [23] propose a taxonomy for backdoors. They state that there are three major types of backdoor: system backdoors, which involve either a single dedicated process which compromises a system, cryptographic backdoors, which compromise cryptographic algorithms, and application backdoors, which they state are versions of legitimate software modified to bypass security

mechanisms under certain conditions. The authors also provide strategies for manual detection of specific types of application backdoor within source code.

Current (semi-)automated backdoor detection methods rely on detecting specific functionality that is associated with triggering backdoor behaviour. Firmalice [19], is a tool developed to detect backdoors within embedded device firmware. The authors propose a model for a class of backdoors they coin *authentication bypass vulnerabilities*. They define the notion of a security policy, which denotes a state that a binary reaches that signifies it is in a privileged state. Firmalice detects if it is possible to violate that security policy (i.e., find a path to a privileged state, without passing standard authentication). HumIDIFy [22] uses a combination of machine learning and targeted static analysis to identify anomalous and unexpected behaviour in services commonly found in Linux-based embedded device firmware. Meanwhile, Stringer [21] attempts to locate comparisons with static data that lead to *unique* program functionality; that is, functionality that can only be executed by a successful comparison with that static data. This models the situation of a backdoor trigger providing access to undocumented functionality. Schuster et al. [17] address the problem of backdoor detection in binaries through the use of dynamic analysis. Using a prototype implementation of their approach, they are able to identify a number of "artificial" and previously identified backdoors.

3 Nomenclature & Preliminaries

In this section we outline terms used in the remainder of this article. A *platform* represents the highest level of abstraction of a device that a given backdoor targets. We define a *system* as the highest level of abstraction required to model a given backdoor, within a *platform*. Since a backdoor can be implemented at any level of abstraction of a *platform* it is designed to compromise – for example, as a dedicated program, a hardware component, or embedded as part of another program – we abstract away from such details. To do this, we model an abstract *system* as a finite state machine (FSM).

When considering a backdoor, there are two perspectives to consider a *system* from: that of the entity that implements a backdoor, and that of the end-user, e.g., a general consumer, or a security consultant analysing the *platform*. To model this situation, we consider four versions of the FSM; for any given *system*, the *Developer* FSM (DFSM) refers to the developer's view of the system, the *Actual* FSM (AFSM) refers to the FSM that models a real manifestation of the system, i.e., a program, the *Expected* FSM (EFSM) refers to the end-user's expectations of the *system*, and finally, the *Reverse-engineered* FSM (RFSM), represents a refinement of the EFSM obtained by reverse-engineering the *actual* system; it can include states and transitions not present within the DFSM or AFSM, e.g., in the case of bug-based backdoors, which we address in §4.

Each state of the FSM describing a *system* can be viewed as an abstraction of a particular functionality – which, in turn can be modelled using a FSM. Thus, we view an entire *system* as a collection of sub-systems, which can be visualised in a layered manner – with each layer representing a view of a part of the *system* at an increasing level detail, as in Fig. 1.

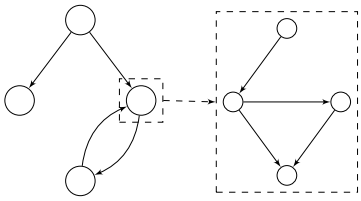

Fig. 1. Multi-layered *system* FSM.

For example, if a given backdoor compromises a router, then we refer to the router as the *platform*. If the backdoor is implemented in software, as a dedicated program, we would view the highest level of abstraction, i.e., the *system*, as the interactions between the processes of the operating system, modelled as a FSM. Each individual running program, or process, can then be modelled by arbitrary levels of FSMs.

3.1 Analysis and Formalisation of FSMs

We specify a FSM as a quintuple: $\theta = (S, i, F, \Sigma, \delta)$, where: S is the set of its states, i is its initial state, F is the set of its final states, Σ is the set of its state transition conditions, e.g., conditional statements that when satisfied cause transitions, and $\delta : S \times \Sigma \rightarrow S$ is its transition labelling function, representing its state transitions.

Inspired by the approach taken by Dullien [15], we view the implemented, or *real* system modelled by a FSM as an emulator for the AFSM of the system. Thus, when the user's EFSM and the AFSM are not equivalent, e.g., the user assumes there is no backdoor present, when there is, specific interactions with the real system will yield unexpected behaviour. How this unexpected behaviour manifests is what determines if that unexpected behaviour means that the system contains a backdoor. Different users of the system will assume different EFSMs. In order to analyse a system, a program analyst, for example, will derive a RFSM – which, for notational ease we refer to as θ_R – by reverse-engineering the *real* system; they do this by making perceptions and observations of its concrete implementation, i.e., the emulator for θ_A. What the analyst will observe is a set of states and state transitions, which are a subset of all those possible within the *platform*, e.g., CPU states. To analyse these states and derive θ_R, the analyst will require a means to map concrete states and transitions of the *platform*, to the level of abstraction modelled by the states and transitions of their FSM. To perform analysis, we assume that an analyst has the following capabilities:

1. They have access to the emulator for the actual FSM (θ_A) – in the case of software, this would be the program binary.
2. They are able to perform static analysis upon the emulator, i.e., using a tool such as IDA Pro, and hence *perceive* a set of system states and state transitions between those states of the real system.
3. They are able to perform dynamic analysis of the system, i.e., with a debugger, and hence *observe* a set of system states and transitions of the real system.

The perceptions and observations of the analyst, along with a means to map concrete states and transitions to abstract FSM states and transitions, allows them to construct a RFSM (θ_R) from the emulator for a AFSM. The granularity of the RFSM will be dependent upon how a system is analysed, e.g., a tool such as IDA Pro will capture components as groups of basic blocks, while components identified in source-code can be represented with a higher-level of abstraction.

3.2 Backdoor Definition

The implementation strategies of backdoor implementers varies widely, therefore, we consider the notion of an abstract backdoor, which we decompose into components. In order to do this, we attempt to answer a number of questions: what is it that makes a set of functionalities, when interacting together manifest as a backdoor? What abstract component parts can be found in all such backdoors? To what extent do we need to abstract to identify all such components?

A distinguishing feature of all backdoors is that they must be triggered. Thus, a pivotal component of any backdoor is its *trigger* mechanism. However, this *trigger* mechanism alone does not constitute a backdoor: what causes it to become active? Another component is needed to account for the satisfaction of the *trigger* condition: i.e., a type of *input source*. Upon *trigger* activation an eventual system state is reached that can be considered the *backdoor-activated* state, which is essentially a state of escalated privileges, privilege abuse or unauthenticated access, i.e., a *privileged state*. To reach this final state, an intermediate component that facilitates the transition from the *normal* system state upon satisfaction of the backdoor *trigger* to the *backdoor-activated* state is required: we refer to this as the backdoor *payload*. Through this reasoning, we show there are four key components that must be present to fully capture the notion of a backdoor. These components are chosen as the minimum set of components required for a backdoor to exist within a system; without the presence of any one of these components the backdoor would not be functional. Using this componentisation, we are able to define a backdoor.

Definition 1. ***Backdoor*** An *intentional* construct contained within a system that serves to compromise its expected security by facilitating access to otherwise privileged functionality or information. Its implementation is identifiable by its decomposition into four components: *input source*, *trigger*, *payload*, and *privileged state*, and the intention of that implementation is reflected in its complete or partial (e.g., in the case of bug-based backdoors) presence within the DFSM and AFSM, but not the EFSM of the system containing it.

3.3 Backdoor Detection

Using Definition 1 as a basis, a backdoor can be modelled as two related FSMs: $\theta_{trigger}$, which represents the *trigger* without a state transition to the *payload*, and $\theta_{payload}$, which represents the *payload* and $F_{payload}$, the set of possible *privileged states*.

Definition 2. ***Backdoor Detection*** A backdoor is detected by obtaining:

$$\theta_R = (S_R, i_R, F_R, \Sigma_R, \delta_R)$$

Within θ_R, the states and transitions of both the *trigger* and *payload* must exist:

$$\Sigma_{trigger} \cup \Sigma_{payload} \subseteq \Sigma_R$$
$$\forall s \in S_{trigger}, \forall \sigma \in \Sigma_{trigger}.\ \delta_{trigger}(s,\sigma) \neq \bot \Rightarrow \delta_R(s,\sigma) = \delta_{trigger}(s,\sigma)$$
$$\forall s \in S_{payload}, \forall \sigma \in \Sigma_{payload}.\ \delta_{payload}(s,\sigma) \neq \bot \Rightarrow \delta_R(s,\sigma) = \delta_{payload}(s,\sigma)$$

The *privileged states* reachable as a result of the *payload* are either final states of θ_R, or states that can be transitioned from to some state of θ_R:

$$S_{trigger} \cup S_{payload} \subseteq S_R$$
$$\forall f \in F_{payload}.\ f \in F_R \vee (f \notin F_R \Rightarrow \exists \sigma \in \Sigma_R.\ \delta_R(f,\sigma) \in S_R)$$

The *payload* must be reachable from the *trigger*, and there must exist a transition to the *trigger* within θ_R:

$$\forall f \in F_{trigger}.\ \exists \sigma \in \Sigma_{trigger}.\ \delta_R(f,\sigma) = i_{payload}$$
$$\exists s \in S_R, \exists \sigma \in \Sigma_R.\ \delta_R(s,\sigma) = i_{trigger}$$

4 A Framework for Modelling Backdoors

In this section we detail a framework for decomposing a backdoor into the four components defined in §3.2; we exhaustively enumerate the types of these components which allows us to both identify and reason about them.

In addition to locating a construct consisting of an *input source*, *trigger*, *payload*, and *privileged state*, to detect a backdoor, an analyst must demonstrate that the construct would be part of the DFSM of the system. For open-source software, this could be done by analysing the source code version control logs, or in closed-source software, analysing the differences between software versions. In other cases, where such analysis is not possible, the following framework can additionally serve as a basis for reasoning about how a backdoor's components can indicate an implementer's intent.

In the proceeding framework, we refer to the RFSM of an end-user that has analysed a particular *system*. Initially, that user will expect functionality that can be modelled by one FSM (their EFSM), and through their analysis they will learn, or derive another FSM (RFSM) that matches what they have learnt about the *system*. Therefore, to discover a backdoor through analysis of the emulator for the AFSM, the RFSM (post-analysis) will contain a backdoor, if there is one present in the AFSM, and they are able to identify it.

During the analysis process, new states and state transitions will be added to the RFSM. We divide these states and state transitions into two categories: those that are explicit, which we say are *discovered* (and always exist within the AFSM) and those that are not explicit, which we say are *created* (and may not exist within the AFSM). To serve this distinction with an example, suppose we have a RFSM that models a program. The explicit states and state transitions that are added to it through analysis are those that represent basic blocks and branches that are *explicitly* part of the program's code (and will always be part of the DFSM and AFSM). Those that are added that are not explicit are in a sense *weird* states and state transitions, which might, for example, be the states representing some shellcode.

4.1 Input Source

If we model the satisfaction of a backdoor *trigger* as a function – `is_triggered` – as in the state machine diagram in Fig. 2, then we can view it as a function that takes at least one parameter (implicit or otherwise) – an *input source* – which is used to decide which state transition that is made as a result of executing that function.

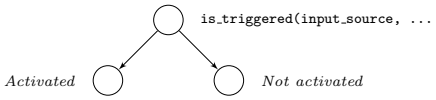

Fig. 2. Idealised Backdoor Trigger.

The value yielded by the *input source* may be derived from any number of inputs to the FSM: it could be a string input by the attacker wishing to activate the backdoor *trigger*, or it could be the value of the system clock such that during a specific time period the backdoor *trigger* becomes active. For this reason we choose to abstract away from the exact implementation details and use the term "*input source*" to represent this component of the backdoor. Note that the *input source* is not the value that causes the activation of the backdoor *trigger*, but rather describes the origin of that input: e.g., a socket or standard input.

4.2 Trigger Mechanism

The backdoor *trigger*, under the correct conditions, will cause the execution of the backdoor *payload*, which will subsequently elevate the privileges of the attacker. We model the backdoor *trigger* as a boolean function where its positive outcome, i.e., when it outputs *true*, will cause a state transition to the backdoor *payload*. The way the FSM transitions to the *payload* as a result of the satisfaction of the *trigger* conditions can be modelled exhaustively with two cases:

1. The state transition is explicit, hence will always exist within the backdoor implementer's DFSM. The backdoor *trigger* is added to the RFSM by adding the explicit states and transitions related to satisfying the backdoor *trigger* conditions, and adding one or more transitions to the *payload*, where those transitions are *discovered* (not newly created) as part of the analysis.
2. The state transition is not explicit. The *trigger* is added to the RFSM by adding explicit states and state transitions related to satisfying the backdoor *trigger* conditions, and by *adding* one or more state transitions that transition to the *payload*, where those transitions are newly created as part of the analysis, i.e., they are not explicit.

To visualise these cases, we use concrete examples in which we use a *system* that is a single program, where the backdoor is embedded as part of the program.

In the first case, we view a *trigger* that is obvious and explicit, where the backdoor is encoded within a single function of the program. This case is shown in Fig. 2. The backdoor *trigger* is comprised of the single state required to satisfy the backdoor *trigger* conditions, i.e., the one labelled `is_triggered(...)`,

```
bool vulnerable_auth_check(
  const char *user, const char *pass) {
  char buf[80], hash[32];

  strcpy(buf, user); strcat(buf, pass);
  create_user_pass_hash(hash, buf);

  return check_valid_hash(hash);
}
```

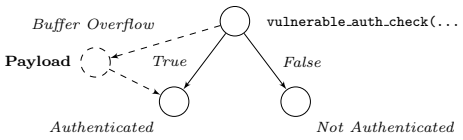

Fig. 3. Bug-based backdoor *trigger*.

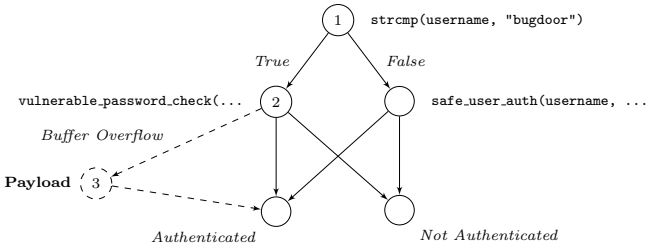

Fig. 4. Hybrid bug-based backdoor *trigger*.

and the state transition to the *Activated* state. In a more realistic scenario, the backdoor *trigger* mechanism may require satisfaction of multiple branch conditions and/or execution of multiple basic blocks and might be obfuscated. Irrespective of these implementation details, the core concept is the same: the collection of checks can be viewed as a single function, whose outcome is used to decide if the backdoor *payload* is transitioned to and hence executed or not, where the transition – a CFG edge in this example – is explicitly part of the FSM.

While the first case considers conditions that are satisfied within a *valid* function CFG, and a transition to the *payload* which is contained entirely within that same *valid* CFG, and thus constitutes *normal* control-flow, the second case of backdoor *trigger* manifests as *abnormal* control-flow. Within a program, we can think of such a construct as akin to a program bug that allows control-flow hijacking. One can conjecture a simple case for this being, a buffer overflow vulnerability, that when exploited correctly, causes a program to transition to a backdoor *payload*, shown in Fig. 3.

Alongside these basic cases, a more complex example of a backdoor *trigger* would be one that relies both on explicit checks and a bug, as visualised in Fig. 4. In this case, a hard-coded credential check against a specific username (`bugdoor`) is used to *guard* access to a vulnerable password check (`vulnerable_password_check`). A username other than `bugdoor` will cause the standard authentication routine (`safe_user_auth`) to be executed, and only a password with a long enough length (and specific content) will lead to the execution of the backdoor *payload*. In this example, the backdoor *trigger* is comprised of the explicit states 1 and 2, and the non-explicit state transition between states 2 and 3, i.e., the *payload* state.

Note that to make the case that all vulnerabilities are backdoor *trigger* mechanisms is a false oversimplification, as such a simplification does not differenti-

ate between accidental and intentional program bugs. We discuss the difficulties present when reasoning about backdoors that are bug-based in §5.

4.3 Payload

A backdoor *payload* can be viewed as the solution to a puzzle: i.e., how to reach a *privileged state* from successfully satisfying the conditions of a *backdoor trigger*. In our model, we represent this by the state transition taken in order to reach a *privileged state*, and any additional states and state transitions that perform prerequisite computation following activation of backdoor *trigger*. In practice, a *payload* component can take many forms, however we can exhaustively categorise all types of *payload* by how they are modelled as part of a RFSM, and how they are transitioned to:

1. The transition to the *payload* is explicit, and does not permit the creation of new states and state transitions (Fig. 5). The *payload* is added to the RFSM by adding explicit states and transitions required to reach a *privileged state*, where those states and transitions are *discovered* by analysis (explicit). They will be contained in the backdoor implementer's DFSM.
2. The transition to the *payload* is explicit, but state(s) reachable due to this transition permit the creation of new states and transitions, e.g., a system that contains an intentional interpreter which can be accessed via a backdoor (Fig. 6). The *payload* is added to the RFSM by adding *discovered* (explicit) states and transitions – which exist in the backdoor implementer's DFSM – from which both newly *created* (non-explicit) and *discovered* (explicit) states and transitions can be reached, which facilitate the eventual transition to a *privileged state*. The non-explicit states and transitions added will not exist within the backdoor implementer's DFSM.
3. The transition to the *payload* is not explicit (bug-based), and the *payload's* states and transitions will either be explicit or non-explicit, e.g., a ROP-based construct. The *payload* is added to the RFSM by adding both newly *created* (non-explicit) and *discovered* (explicit) states and transitions, which facilitate the transition to a *privileged state*. The non-explicit states and transitions added will not exist within the backdoor implementer's DFSM.

4.3.1 Payload examples

To give concrete examples of the variants of backdoor *payload*, we once again demonstrate backdoors that are implemented within programs.

Explicit transition to payload with explicit payload components This class of *payload* (case 1 above) is inherently an intentional construct and requires no abnormal control flow for it to be executed. An example of a backdoor with such *payload* is shown in Fig. 5. The backdoor *trigger* condition (state 1) is a hard-coded credential check, which if satisfied, will transition to the backdoor *payload* (transition from state 1 to 2). In the *payload*, the backdoor user's permissions are first elevated (state 2) and then a shell is opened for that user (state 3), which allows them to transition to the *privileged state* (state 4).

```
/* Trigger; if active then: (1) -> (2) */
if (strcmp(user._name, "backdoor") == 0) {
  /* Payload */
  user._is_admin = true;  // (2)

  /* Transition to privileged state */
  open_shell(&user); // (3) -> (4)
}
```

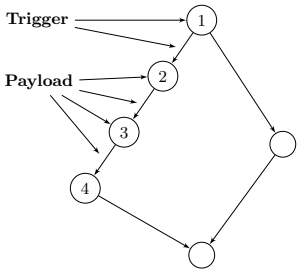

Fig. 5. Explicit transition to *payload*, where *payload* has explicit components.

```
/* Trigger; if active then: (1) -> (2) */
if (strcmp(req._path, "/BKDRLDR") == 0) {
  /* Payload; req._data == payload input */
  run(&req._data); // (2) -> (3)
}
```

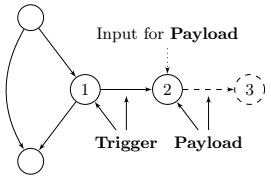

Fig. 6. Explicit transition to *payload* with both explicit and non-explicit components.

Explicit transition to payload with explicit and non-explicit payload components In this case (case 2 above), we model a backdoor that enables an attacker to perform computation not part of the developer's DFSM, without being in a state that is bug-induced. An example of such a backdoor is shown in Fig. 6; if the backdoor *trigger* is satisfied, the program will interpret and execute an input supplied by the user of the backdoor. The *trigger* condition is a check to see if a user is requesting access to a specific path (state 1), if it is, then the *payload* is transitioned to (state 1 to 2), where the data sent with the request (`req._data`) is used as input to an interpreter (state 2, via `run`). In this case, the *privileged state* (state 3) transitioned to is dynamically constructed as a result of the input to the interpreter executed in state 2.

```
void some_function() {
  char buf[80];
  /* ... */
  /* Backdoor activated if len(input)
     causes buffer overflow */
  strcpy(buf, input); // (1) -> (2)
  return;
}

void other_function() {
  /* ... */
  /* Payload reaches via (2) -> (3) */
  g_user._is_admin = true; // (3)
  open_control_panel(); // (4)
}
```

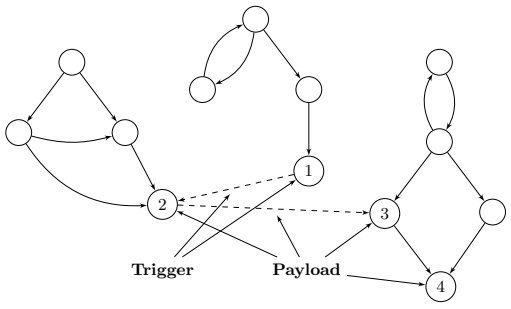

Fig. 7. Non-explicit transition to *payload*, where *payload* has both explicit and non-explicit components.

Non-explicit transition to payload with explicit and non-explicit payload components In the final case (case 3 above), we model backdoors that

have a *trigger* mechanism that is bug-based, i.e., allows an attacker to perform computation not part of the developer's DFSM. We visualise such a case in Fig. 7; here the *trigger* consists of an *intentional* buffer overflow bug in `some_function` (state 1), which if exploited – in this case with a ROP-based *payload* – transitions (via 1 to 2) to the *payload*. The *payload* consists of states 2 and 3, and the transitions from states 2 to 3, and 3 to 4. As a result of the *payload*, the user is granted administrative privileges (state 3), and entered into a (privileged) control panel via `open_control_panel` in `other_function` (state 4).

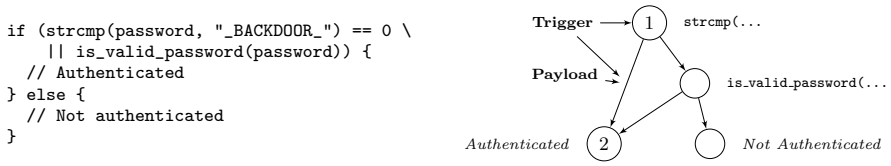

Fig. 8. A backdoor *payload* composed solely of a state transition.

Single transition payloads We note there is a special case for both cases 1 and 3, namely, where the *payload* is composed of only a single state transition. That is, no additional computation is undertaken as part of the *payload*, rather the *payload* shares its state transition with the backdoor *trigger*, as shown in Fig. 8. This special case accounts for situations where the backdoor *trigger* acts like a trapdoor (state 1), allowing an attacker to bypass a (potentially) more complex check for user-authentication, and rather provides a direct transition to a *privileged state* (the transition from state 1 to 2). The form of the *payload* is identical for cases 1 and 3, other than the explicitness of the state transition (the *payload*) between the *trigger* and the *privileged state*.

4.3.2 Payload obfuscation

So far, we have not considered how a backdoor implementer might hide a backdoor's presence – other than by using a bug-based *trigger* mechanism. While such a trigger is simple to implement, it offers the implementer no control over how the backdoor will eventually be used; this control can be regained, by for example, limiting the computational freedom of newly created states. In this section we explore how a backdoor implementer can obfuscate *payload* components.

Since backdoor *payloads* that contain only explicit states and state transitions are obvious and thus, intentional constructs, an obfuscated *payload* by nature must be implemented through the use of some degree of abnormal control flow, i.e., non-explicit states and state transitions. An example of such a *payload* is one *derived* by reusing components of the system it is implemented within to obscure its execution, e.g., for a program, from static analysis methods. From an attacker's perspective, the only way to execute such a backdoor is either to have prior knowledge of the *payload*, or *solve a puzzle* and derive it from the original system. Andriesse et al. [14] describe such a backdoor (examined in further detail

in §6), whereby its *payload* component is composed of multiple code fragments embedded and distributed throughout a binary which execute in sequence upon the backdoor being triggered. Fig. 7 shows a naïve example such a *payload*.

Another example is that where a *payload* can be derived from attacker controlled data. In the simplest case, this is akin to shellcode often executed as a result of successful exploitation of a buffer overflow vulnerability: it shares a commonality that it doesn't rely upon any existing program components. In more sophisticated cases, such a backdoor *payload* might take a hybrid approach: where either user-data is interpreted by the program itself, or components of the program are used alongside the user input. Fig. 6 shows a simple example such a *payload*. In both of these examples, the *payload* components are implemented in a so-called *weird machine* as defined by Oakley and Bratus [16].

4.4 Privileged State

Following successful activation of the backdoor *trigger* and subsequent transitioning from the associated *payload*, the system will enter into a *privileged state*. There are two possibilities for this state: either it can be reached under *normal* system execution, or it can only be reached through activation of the backdoor. If we consider *privileged states* by how they are added to a RFSM, then one that is newly created, i.e., is non-explicit, will not be reachable under normal system execution, meanwhile, one that is explicit, may or may not be reachable under normal execution: for example, while the *privileged state* might be explicit, the only way to reach it might be via the backdoor *trigger*.

In the case of a *privileged state* reachable through normal execution, consider the backdoor presented in Fig. 8, which models a hard-coded credential check. The *privileged state* (state 2) of the backdoor is both reachable via the backdoor *trigger* (from state 1), and the state labelled `is_valid_password`.

For the other case, where the *privileged state* is not reachable by a *legitimate* user, it is essentially *guarded* by the activation of the backdoor. This case can further be sub-categorised. The first variant is where the *privileged state* is explicit, as in Fig. 5; the *privileged state* (state 4) is only reachable through activation of the backdoor *trigger* (state 1 and the transition from state 1 to state 2). In this example, the *privileged state* manifests as an undocumented *backdoor* shell, where after entering a specific username, the attacker is able to perform additional functionality, not otherwise possible. The other variant is a *privileged state* that provides an attacker access to functionality that is not available to a *legitimate* user, where that functionality does not explicitly exist within the system – as shown in Fig. 6. Here the *privileged state* (state 3) is some function of attacker input, i.e., the result of `run(&req._data)`.

5 Practical Detection & Deniability

Backdoor detection in practice will happen through, e.g., manually reverse-engineering a program binary or observing a backdoor's usage through suspicious system events, such as anomalous network traffic. As is, our proposed framework oversimplifies as it doesn't model intention. If we knew that a particular vulnerability was placed *intentionally*, then there would be no question

that the vulnerability was placed deliberately to act as a backdoor. Thus, in this section we answer the question: if we have identified a backdoor-like construct, can we distinguish it from an accidental vulnerability, and if so, how deniable is it?

In order to make such a distinction, recall that we can view a system from four perspectives: its DFSM, AFSM, EFSM, and RFSM. If a backdoor-like construct has been identified, then it will be present in both the emulator for the AFSM and the RFSM. To state that the construct is a backdoor – and was placed intentionally – we must show that it, or some part of it was present within the DFSM. In some cases, the intent is explicit and hard-coded in the implementation – i.e., it leaves no ambiguity. The most obvious example of this is a hard-coded credential check which serves to bypass standard authentication. Indeed, all cases of backdoor that transition explicitly, i.e., discoverable by analysis, from the satisfaction of their *trigger* conditions to their *payload* can be considered intentional.

In the other case, where that transition is non-explicit, i.e., bug-based, various approaches can be taken. For instance, in the case of software, where version control logs are available, it is possible to identify the exact point where a backdoor has been inserted as well as its author (e.g., the failed attempt to backdoor the Linux kernel in 2003 [1]). For binary-only software, where there exists multiple versions of that software, it is possible to identify the version the backdoor was introduced in, and reason about its presence by asking the question *was there a legitimate reason for making such a change to the software?* Further, we can consider the explicitness of the backdoor components: for example, if a code fragment exists within a binary that does nothing more than facilitate privilege escalation, and it is unreachable by normal program control-flow, then there is an indication of intent. A similar case can be made if the satisfaction of the *trigger* conditions rely on checks discoverable by analysis, as well as a bug. Unfortunately, all of these approaches have non-technical aspects and rely on human intuition – thus, do not provide a concrete proof of intent. We are therefore left with three possible ways to classify backdoor-like constructs:

Definition 3. ***Intentional backdoor*** Those constructs that can be unambiguously identified as backdoors: the transition from their *trigger* satisfaction to their *payload* is explicit. Will be present in the DFSM, AFSM, and if found, the RFSM, but not the EFSM.

Definition 4. ***Deniable backdoor*** Those constructs that fall into a grey area, where the transition from their *trigger* satisfaction to their *payload* is non-explicit (i.e., it appears to be a bug), but from a non-technical perspective can be argued to be intentional. Will be present in the AFSM, if found, the RFSM, but not the EFSM; we cannot definitively tell if it is in the DFSM.

Definition 5. ***Accidental vulnerability*** Those constructs where there is no evidence – technical, or otherwise – to suggest any intent, and the transition from their *trigger* satisfaction to their *payload* is non-explicit. Will be present in the AFSM, and if found, the RFSM, but not the DFSM or EFSM.

From a purely technical perspective, a *deniable* backdoor will be indistinguishable from an *accidental* vulnerability. Consider, for example, a simple buffer overflow vulnerability and its corresponding exploit. If this vulnerability was deliberately placed then it is a backdoor, otherwise it is just a vulnerability coupled with an exploit. As we do not know anything about the implementer's intention we cannot discern between the two. Thus, a vulnerability can be seen as an unintentional way to add new state transitions, or states to a system's FSM, while an exploit is a set of states and state transitions such that when combined with a vulnerability within a given FSM, provides a means to compromise the believed security of the system modelled by that FSM. In contrast to backdoors and vulnerabilities, a construct providing standard privileged access will be intentional and manifest within the DFSM, AFSM, EFSM, and RFSM of a system.

6 Discussion & Case-studies

In order to demonstrate our framework, we provide a number of case studies. We show examples from both the literature and real-world backdoors, which have been detected manually. For each backdoor, we reason about if and why its implementation can be considered deniable in respect to our definitions and analyse it by performing a complete decomposition of its implementation using our framework. Finally, we provide a discussion of how our framework can be used to reason about methods for detecting backdoors.

Table 1 shows eleven real-world backdoors, each decomposed using our framework. As each backdoor can be modelled with explicit states and state transitions, by definition 3, none are deniable, thus, their implementers should be held accountable. The remainder of this section provides case-study of a complex, deniable (by definition 4) backdoor.

Nginx Bug-Based Backdoor Andriesse and Bos [14] describe a general method for embedding a backdoor within a program binary. Their technique utilises a backdoor *trigger* based upon an intentional program bug combined with a hard-coded *payload* composed of intentionally misaligned instruction sequence fragments. Their *payload* is, in a sense, obfuscated, yet fixed; its implementation exploits the nature of the x86 instruction set, whereby byte sequences representing instructions can be interpreted differently when accessed at different offsets.

The authors demonstrate their approach by modifying the popular web-server, Nginx, and embedding a remotely exploitable backdoor. In their implementation, a would-be attacker provides a crafted input, which serves to satisfy the backdoor *trigger* conditions; this input is provided as a malformed HTTP packet – the *input source* will therefore be a network socket. Fig. 9 provides a code listing adapted from [14] which contains the backdoor *trigger* conditions. Those conditions are: `have_err == 1`, and `err_handler != NULL`, which are set as a result of the use of uninitialised variables `have_err` and `err_handler` in the `ngx_http_finalize_request` function, which take the values of `badc` and `hash` in `ngx_http_parse_header_line`. The bug manifests due to the fact the two functions stack frames overlap between their invocations. The intended *pay-*

Table 1. Real-world backdoors modelled using framework.

Backdoor description	Input source	Trigger	Payload	Privileged State
D-Link router backdoor "Joel's backdoor" [5]; bypass standard authentication by setting a specific user-agent when accessing web-configuration interface.	Network socket	`strcmp(ua, "xmlset_roodkcableoj28840ybtide") == 0`	Trigger conditions satisfied: explicit transition by matching with user-agent	Authenticated as legitimate user
Tenda router backdoor [2]; additional UDP server embedded within web-server allowing remote command execution.	Network socket; UDP port 7329	Correct packet format: `strcmp("w302r_mfg", packet->magic) == 0` and `packet->command_byte == 'x'`	Trigger conditions satisfied; arbitrary command executed via: `popen(packet->command, "r")`	Dependent upon `packet->command` payload input
TCP-32764 router backdoor [6]; multiple vendors (Netgear, Cisco, Belkin, ...); remotely accessible undocumented service allows modification of configuration (e.g. device credentials), and command execution.	Network socket; TCP port 32764	Correct packet format: `"ScMM" | 7 | cmd_len | cmd`	Trigger conditions satisfied (correct packet format with); executes `cmd` via `popen`	Dependent upon `cmd` payload input
Quanta LTE router backdoor [11]; dedicated UDP service "appmgr" if sent specific string enables an unauthenticated root shell via Telnet.	Network socket; UDP port 39889	`strncmp("HELODBG", data, 7) == 0`	Trigger conditions satisfied: `data` matches `HELODBG`; starts Telnetd as root using: `system("/sbin/telnetd -l /bin/sh")`	Shell accessible via TCP port 23
Sony IP camera backdoor [10]; combination of HTTP request with specific values set as parameters and hard-coded HTTP authentication credentials, can start Telnet service on device remotely via web-server.	Network socket; TCP port 80/443	HTTP request to `/command/prima-factory.cgi` with parameters that satisfy `strcmp("cPoq2fi4cFk", param1) == 0` and `strcmp("zKw2hEr9", param2) == 0`; hard-coded username and password for HTTP authentication: `strcmp("primana", username)` == 0 and `strcmp("primana", password) == 0`	Trigger conditions satisfied; performs `system("/usr/sbin/inetd")` which starts `telnetd` using configuration in `/etc/inetd.conf`	Shell accessible via TCP port 23
RaySharp DVR backdoor [21]; hard-coded credentials bypass standard authentication in web-interface.	Network socket; TCP port 80/443	`strcmp("root", username) == 0` and `strcmp("519070", password) == 0`	Trigger conditions satisfied: explicit transition by matching credentials	Authenticated as root/administrative user
Western Digital My Cloud NAS backdoor [13]; setting specific value in (unencrypted) cookie when accessing web-interface allows user to login as administrator.	Network socket; TCP port 80/443	`$_COOKIE["isAdmin"] == 1`	Trigger conditions satisfied; explicit transition via value being set correctly	Authenticated as administrative user
Q-See DVR backdoor [21]; multiple hard-coded username/password combinations, each combination gives access to additional functionality, as well as bypassing standard authentication.	Network socket/virtual keyboard	`strcmp(username, "admin") == 0` and `strcmp("6036huanyuan", password) == 0`; multiple possible passwords	Explicit transitions and states dependent upon password	Depends on password; authenticated as administrative user with greater privileges – alternate "control-panel"
D-Link router backdoor [4]; execution of arbitrary operating system commands through unauthenticated PHP script.	Network socket; TCP port 80/443	POST request to `command.php`, with `cmd` parameter equal to command to run	Trigger conditions satisfied; explicit transition if parameter is set; executes command specified by `cmd`	Dependent upon payload input
Netis router backdoor [12]; custom network service (`igdmptd`) protected with a hard-coded password; enables (among other functionality) execution of arbitrary commands.	Network socket; TCP port 53413	Authenticate to service using hard-coded credentials: `strcmp("netcore", password) == 0`	Trigger conditions satisfied; explicit transition to custom command shell; input arbitrary commands executed using `popen`	Dependent upon commands entered for payload input
3S Vision N5072 camera backdoor [19]; hard-coded credential check for HTTP authentication, bypasses standard authentication.	Network socket; TCP port 80/443	Authenticate to service using hard-coded credentials: `strcmp(username, "3sadmin") == 0` and `strcmp(password, "27988303") == 0`	Trigger conditions satisfied; explicit transition to authenticated state	Authenticated as legitimate user of device

```
ngx_int_t ngx_http_parse_header_line(/* ... */) {
  u_char badc; /* last bad character */
  ngx_uint_t hash; /* hash of header, same size as pointer */
  /* ... */
}

void ngx_http_finalize_request(ngx_http_request_t *r, ngx_int_t rc) {
  uint8_t have_err; /* overlaps badc */
  void (*err_handler)(ngx_http_request_t *r); /* overlaps hash */
  /* ... */
  if(rc == NGX_HTTP_BAD_REQUEST && have_err == 1 && err_handler) {
    err_handler(r); /* points to hidden code, set by trigger */
  }
}

void ngx_http_process_request_headers(/* ... */) {
  rc = ngx_http_parse_header_line(/* ... */);
  /* ... */
  ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST); /* bad header */
}
```

Fig. 9. Source-code listing for Nginx backdoor *trigger*.

load states are meant to be those embedded as *weird* states, however additional states are possible, for example, if the attacker provides a different input packet to that expected by the implementers. The *privileged state* depends on the backdoor *payload*. We visualise the backdoor in Fig. 10; the *trigger* is captured by state 1 and the non-explicit, bug-based transition to state 2; the *payload* consists of state 2 and the transition between state 2 and 3; state 3 is the *privileged state*.

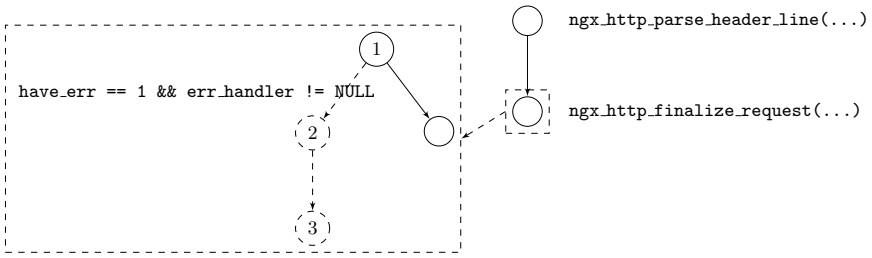

Fig. 10. Multi-layered FSM for Nginx backdoor.

From a technical standpoint the backdoor is deniable (by definition 4), this is due to its *trigger* transition being bug-based, whilst its *payload*, if discovered, is arguably intentional. The componentisation using our framework allows us to visualise a complex backdoor succinctly, which would otherwise be buried across multiple functions in thousands of lines of source code. Further, its componentisation allows us to reason about how such a backdoor can be detected: for example, we could attempt to detect its bug-based trigger condition using symbolic execution; alternatively, we could heuristically attempt to identify its *payload* by scanning for misaligned instruction sequences that branch to other instruction sequences of the same kind, where the combination of those sequences would serve to elevate an attacker's privileges.

Table 2. Tool detection methodology decomposed using framework.

Tool	Input source	Trigger	Payload	Privileged State
Firmalice [19]	Partial	Partial	No	Partial
HumIDIFy [22]	Partial	No	Partial	No
Stringer [21]	No	Partial	Partial	No
Weasel [14]	No	Partial	Partial	Partial

6.1 Backdoor Detection Methodologies

Our framework provides not only a means to reason about backdoors, but also backdoor detection techniques. Table 2 shows the decomposition of the detection methodologies of four state-of-the-art backdoor detection tools. Each tool claims to detect a particular subset of backdoor types. However, while these tools are all effective, none consider a complete model of backdoors, and, as a result, are limited in their effectiveness.

Firmalice [19] is designed to detect authentication bypass vulnerabilities. It uses a so-called security policy to define the observable side-effects of a program being in a *privileged state*. Using a specified *input source*, it attempts to find data provided via this *input source* that satisfies the conditions – i.e., akin to a backdoor *trigger* – required to observe the side-effects specified by the security policy. Firmalice has no notion of a *payload state*; when entered, a *payload state* might leave a program in a *privileged state* that is not captured by a given security policy, for instance, where the *privileged state* reached by a backdoor user is different from that of a legitimate user reaches, e.g., the Q-See DVR backdoor from Table 1. Firmalice is able to detect such a *privileged state* by modification of the input security policy, however, to do so will require the same amount of manual analysis to detect the entire backdoor as it would to identify the *privileged state*.

HumIDIFy [22] aims to detect if a program can execute functionality it should never execute under normal circumstances. This might be the establishment of a suspicious *input source*, or the execution of API that is considered anomalous, i.e., what might be part of a backdoor *payload*. However, since it does not consider the notion of a *trigger*, it is unable to distinguish between *abnormal* program behaviour that is benign – because it can only be performed by a legitimate user, and behaviour that is genuinely anomalous – that is part of a backdoor. Again, this is due to their approach not considering a complete model of a backdoor.

Stringer [21] attempts to detect static data used as program input that is responsible for either enabling authentication bypass vulnerabilities, or used for triggering the execution of undocumented functionality. To do this it uses a scoring metric, which ranks static data, that when matched against, leads to the execution of unique functionality, i.e., functionality not reachable by other program paths. Stringer considers the partial notion of a backdoor *trigger* and uses heuristics for identifying *payload-like* constructs. It does not consider the notion of *input source*, or *privileged states*, and as a result of the latter, is unable to meaningfully score data that leads to states that are actually privileged higher than those that are not.

Weasel [14] detects both authentication bypass vulnerabilities and undocumented commands in server-like program binaries. It works by attempting to

automatically identify so-called deciders (akin to backdoor *triggers*) and handlers (akin to the combination of a backdoor *payload* and *privileged state*) which then serve to aid in detection of backdoors. Their approach does not fully model the notion of a backdoor; it does not consider an *input source* at all, rather, the approach models a single input for the program, and data from that source, when processed, is assumed to reveal all deciders and handlers. The Tenda webserver backdoor in Table 1 acts as an undocumented command interface, its *input source* is a UDP port; in this case, the backdoor uses a separate *input source* from the standard input to the program, i.e., TCP port 80 or 443. Since Weasel does not capture the notion of an *input source*, it will be unable to detect such a backdoor – not due to a deficiency in its detection method, but because it does not consider a complete model of a backdoor.

7 Future work

Our framework does not intend to provide a direct means to detect backdoors, rather it serves as a general means to decompose backdoors in an abstract way. In §6.1, we discuss concrete implementations of detection methodologies; in each case we are able to highlight deficiencies in those methods due to them not fully capturing the rigorous definition of a backdoor, as outlined in this work. Thus, a backdoor detection methodology based upon our proposed framework would be a natural extension of this work. Further, while our formalisations attempt to capture any backdoor-like functionality, backdoors introduced into a system by, e.g., a deliberate side-channel vulnerability would prove difficult to model using our FSM-based abstraction; we view this as an additional area for investigation.

8 Conclusion

In summary, we have provided a definition for the term backdoor, definitions for backdoor detection, deniable backdoors, and a means to discern between intentional backdoors and accidental vulnerabilities. We have presented a framework to aid in identifying backdoors based upon their structure, which also serves as a means to compare existing backdoor detection approaches, and as a basis for developing new techniques. To demonstrate the effectiveness of our approach, we have analysed twelve backdoors of varying complexity. In each case, we have been able to concisely model those backdoors, which previously, might have manifested as hundreds or thousands of assembly language instructions in a disassembler. We have used our framework to evaluate four state-of-the-art backdoor detection approaches, and in all cases, have shown that none consider a complete model of backdoors, and, as a result, their potential effectiveness is limited.

References

1. An attempt to backdoor the kernel, 2003. https://lwn.net/Articles/57135/.
2. From China with Love, 2013. http://www.devttys0.com/2013/10/from-china-with-love/.
3. How a Crypto 'Backdoor' Pitted the Tech World Against the NSA, 2013. https://www.wired.com/2013/09/nsa-backdoor/.
4. Multiple Vulnerabilities in D-Link DIR-600 and DIR-300 (rev B), 2013. http://www.s3cur1ty.de/node/672.

5. Reverse Engineering a D-Link Backdoor, 2013. http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/.
6. TCP-32764 Backdoor, 2013. https://github.com/elvanderb/TCP-32764.
7. Why everyone is left less secure when the NSA doesn't help fix security flaws, 2013. https://bit.ly/2JJ9Zsg.
8. Inside the EquationDrug Espionage Platform, 2015. https://securelist.com/inside-the-equationdrug-espionage-platform/69203/.
9. Adups Backdoor, 2016. https://www.kryptowire.com/adups_security_analysis.html.
10. Backdoor in Sony IPELA Engine IP Cameras, 2016. https://sec-consult.com/en/blog/2016/12/backdoor-in-sony-ipela-engine-ip-cameras/.
11. Multiple vulnerabilities found in Quanta LTE routers, 2016. http://pierrekim.github.io/blog/2016-04-04-quanta-lte-routers-vulnerabilities.html.
12. Netis Router Backdoor Update, 2016. https://blog.trendmicro.com/netis-router-backdoor-update/.
13. Hacking the Western Digital MyClound NAS, 2017. https://blog.exploitee.rs/2017/hacking_wd_mycloud/.
14. D. Andriesse and H. Bos. Instruction-Level Steganography for Covert Trigger-Based Malware. *11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2014.
15. T. F. Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, Preprint, 2017.
16. J. Oakley and S. Bratus. Exploiting the Hard-working DWARF: Trojan and Exploit Techniques with No Native Executable Code. *5th USENIX Conference on Offensive Technologies*, 2011.
17. F. Schuster and T. Holz. Towards Reducing the Attack Surface of Software Backdoors. *2013 ACM SIGSAC conference on Computer & Communications Security*, 2013.
18. R. Shapiro, S. Bratus, and S. W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. *7th USENIX Conference on Offensive Technologies*, 2013.
19. Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. *2015 Network and Distributed System Security Symposium*, 2015.
20. S. J. Tan, S. Bratus, and T. Goodspeed. Interrupt-oriented Bugdoor Programming: A Minimalist Approach to Bugdooring Embedded Systems Firmware. *30th Annual Computer Security Applications Conference*, 2014.
21. S. L. Thomas, T. Chothia, and F. D. Garcia. Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality. *22nd European Symposium on Research in Computer Security*, 2017.
22. S. L. Thomas, F. D. Garcia, and T. Chothia. HumIDIFy: A Tool for Hidden Functionality Detection in Firmware. *14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.
23. C. Wysopal and C. Eng. Static Detection of Application Backdoors. *Black Hat USA*, 2007.
24. J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas. Implementation and Implications of a Stealth Hard-drive Backdoor. *29th Annual Computer Security Applications Conference*, 2013.
25. Y. Zhang and V. Paxson. Detecting Backdoors. *9th USENIX Conference on Security Symposium*, 2000.