2012 IEEE Symposium on Security and Privacy

A Framework to Eliminate Backdoors from Response-Computable Authentication

Shuaifu Dai[1], Tao Wei[1,2*], Chao Zhang[1], Tielei Wang[3], Yu Ding[1], Zhenkai Liang[4], Wei Zou[1]

[1]Beijing Key Lab of Internet Security Technology
Institute of Computer Science and Technology
Peking University, China

[2]University of California, Berkeley
[3]College of Computing, Georgia Institute of Technology
[4]School of Computing, National University of Singapore

***Abstract*—Response-computable authentication (RCA) is a two-party authentication model widely adopted by authentication systems, where an authentication system independently computes the expected user response and authenticates a user if the actual user response matches the expected value. Such authentication systems have long been threatened by malicious developers who can plant backdoors to bypass normal authentication, which is often seen in insider-related incidents. A malicious developer can plant backdoors by hiding logic in source code, by planting delicate vulnerabilities, or even by using weak cryptographic algorithms. Because of the common usage of cryptographic techniques and code protection in authentication modules, it is very difficult to detect and eliminate backdoors from login systems. In this paper, we propose a framework for RCA systems to ensure that the authentication process is not affected by backdoors. Our approach decomposes the authentication module into components. Components with simple logic are verified by code analysis for correctness; components with cryptographic/obfuscated logic are sandboxed and verified through testing. The key component of our approach is *NaPu*, a native sandbox to ensure pure functions, which protects the complex and backdoor-prone part of a login module. We also use a testing-based process to either detect backdoors in the sandboxed component or verify that the component has no backdoors that can be used practically. We demonstrated the effectiveness of our approach in real-world applications by porting and verifying several popular login modules into this framework.**

I. Introduction

User authentication is the basis of access control and auditing. Through the login process, the authentication system verifies a user's identity to grant the user proper privilege in the system. The important role of a login module makes it an attractive target for attackers. A common type of login module attacks is through backdoors: malicious developers intentionally leave code in a login module to bypass normal authentication, allowing them to get unauthorized privilege. There have been many incidents where insiders left backdoors in login modules to gain unauthorized access to computer systems, e.g., the backdoor in Cart32 Shopping Cart [5].

*Corresponding author. Email: *wei_tao@pku.edu.cn*

Due to the complexity of large software systems, it is very hard to completely detect this type of threats.

Based on how the authentication system interacts with users, it typically falls into two categories: after a user responds to the authentication challenge, the authentication system either compares the user's response with an expected value computed from known credentials of the user, or uses the user's response as part of a complicated authentication computation. The authentication system of the first type computes the expected response value using credentials on the server. The user is authenticated if the computed value matches the user's response. We refer to this type of authentication as *response-computable authentication (RCA)*. In the other type of authentication systems, the user response is used as an input to the authentication computation, which is based on techniques such as public-key cryptography [26] and zero-knowledge proof [6]. In this paper, we focus on backdoors in the first type of authentication system, response-computable authentication (RCA), which is widely used in authentication systems.

Malicious developers have various ways to plant and hide backdoors in RCA login modules. The simplest way is to add a hard-coded username/password pair. Alternatively, a backdoor condition can be an obfuscated relationship between the username and password. Besides user inputs, a backdoor may also be triggered by system events or internal state of the authentication process. Moreover, malicious developers can exploit a vulnerability in the login module to circumvent the normal authentication process. They can also embed cryptographic backdoors in RCA modules, e.g., using a carefully constructed insecure hash function to make two responses collide at a controllable probability to allow the backdoor's creator to login.

Although it is straightforward to detect hard-coded username/password pairs, it is very difficult to use program analysis to find backdoors in login modules that involve heavy cryptographic computations or code protection and obfuscation. Manual source code review is also difficult to detect cryptographic backdoors. Furthermore, a malicious developer can directly instrument

DOI 10.1109/SP.2012.10

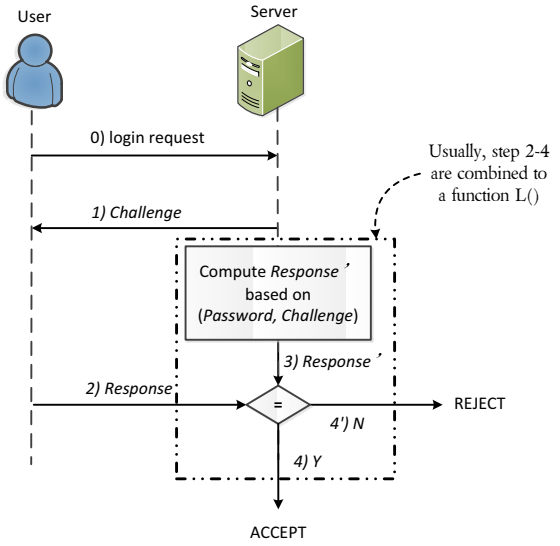

Figure 1. Typical steps in Response-Computable Authentication (RCA).

backdoors at the binary level. For example, a trojaned compiler can create a backdoor in the programs it generates, which is based on hard-coded passwords [32].

Figure 1 illustrates the process of a typical two-party login protocol using RCA, based on techniques such as plain text password, CRAM-MD5 [21], RSA SecureID [3], time-based one-time password (TOTP) [4], and HMAC-based one-time password (HOTP) [24]. On receiving the authentication request (step 0), the server generates a challenge to the user (step 1). After it receives the response from the user (step 2), the server computes the expected response value using the credentials on the server (step 3). It authenticates the user if the user's response matches the server's computed response value (step 4). Step 3 and step 4 form the RCA authentication procedure. We use $L()$ to denote the RCA decision function. It maps user response to a boolean value, TRUE for "accept" and FALSE for "reject".

By studying several types of RCA backdoors, we observe that all such systems treat the decision function $L()$ as a blackbox: they authenticate users only by the return value of $L()$, no matter *how this return value is generated*. As a result, they cannot distinguish authentication decisions resulted from authentic user response and those resulted from backdoors. For example, when the $L()$ contains a backdoor using a hard-coded username, it immediately returns true when the username is provided.

In this paper, we design a secure framework for RCA to ensure that the authentication result is not affected by backdoors. Since backdoors in login modules are caused by the blackbox nature of the authentication process, the high-level idea is to decompose the RCA login module into components: for components with simple logic, verify theirs correctness by code review and analysis; for components with cryptographic/obfuscated logic, sandbox them to prevent the logic from being affected by attackers, and then verify the logic through testing.

The decision function $L()$ has two main components, *response computation* (denoted as $f()$) and *response comparison*. The response comparison is a simple components that can be directly verified by code review, but we need to make sure it cannot be bypassed. The *response-computation function* depends on assistant components, such as reading the password database. The assistant components are typically simple and can be checked for backdoors by code review and analysis. The major challenge arises from the response-computation function, which often contains a lot of cryptographic operations, making it very difficult to detect backdoors by code inspection. Our solution ensures that the response-computation function to be a *pure function*, which returns the same result without side effects each time it is invoked on the same set of arguments [14].

Our approach ensures response-computation function to be free of backdoors based on the following observation: for one login try, there is only one correct response which we can explicitly get from the response-computation function. Following this observation, we use formal analysis to identify the upper bound of backdoor usability in a login module, which forms a theoretical basis of our testing methods. Through the testing, either we can detect the possible backdoor or we can ensure the backdoor cannot be used by its creator. To the best of our knowledge, this approach is the first to give formal analysis of probability of authentication backdoor usability.

To achieve this goal, we design *NaPu*, a native sandbox to ensure pure functions, and use it to purify the response-computation function. NaPu has several features: *vulnerability isolation*, *global state isolation*, and *internal state reset*. These features can prevent an attacker from triggering backdoors stealthily. We build the NaPu sandbox based on Native Client (NaCl) [37], and implement our framework to secure RCA modules. We ported several widely used login modules, such as CRAM-MD5, HOTP, TOTP, into this framework, and verified that these ported libraries are backdoor-free. Our results showed that our solution can be easily applied to real-world systems with acceptable performance overhead.

In summary, we develop a solution to ensure that response-computable authentication is free of impacts from backdoors. Our solution either detects hidden

backdoors or ensures the malicious developer cannot bypass authentication even with a backdoor planted in the authentication process. Our solution makes the following contributions:

- We propose a novel framework to guarantee that the decision of an RCA login module is not affected by backdoors. The key idea is to ensure each sub-component of the authentication process free of backdoor influence, either by analysis of simple application logic, or by sandboxing and testing cryptographic/obfuscated logic.
- We give a systematic analysis of authentication backdoors, and formally prove the upper bound of possibility that a backdoor can be used by attackers in an RCA login module.
- We build NaPu, a NaCl-based pure-function-enforcing sandbox, to support the framework.
- We have prototyped our framework and ported many widely-used login modules. Our evaluation verified that they are free of backdoors.

The rest of this paper is organized as follows: Section II introduces RCA backdoors and the adversary model. Section III describes our intuition and approach design. Section IV analyzes the security of our solution. Section V explains our prototype implementation. We present the evaluation results in Section VI. Section VII discusses limitation of our solution. We discuss related work in Section VIII. Section IX concludes this paper.

II. Background

In this section, we describe the adversary model for our approach, i.e., what malicious developers can do and what they cannot to launch an RCA-backdoor attack. Furthermore, we discuss characteristics of backdoors in RCA login modules, and categorize these backdoors. We also introduce the concept of pure function, which is the basis of our solution.

A. Adversary Model

We focus on RCA backdoors in login modules. The backdoors are implemented by malicious developers who have the opportunity to access developing environments and modify code or binaries. These malicious developers include malicious insider developers and even intruders. They can implement backdoors in software during the development process, but they cannot interfere in the deployment environment.

Methods to plant backdoors. Backdoors can be planted in different ways. The straightforward way is to modify the source code directly. For example, in the ProFTPD incident [2], the intruder concealed a backdoor in the source code package. The malicious developer can modify the *development environment*, such as the compiler. Thompson's compiler backdoor [32] is such an example. The malicious developer can also directly modify the binary code to insert a backdoor. The latter two types of backdoor-planting methods cannot be detected by source code review.

Methods to avoid detection. We assume malicious developers cannot use obfuscation or anti-debugging techniques to build backdoors, which are not permitted when code review is required. However, the attacker still has a number of ways to prevent the backdoor from being detected. They may construct a subtle vulnerability that eludes source code inspections, and gain system privilege by a malformed input. They may design and use a *weak cryptographic algorithm* in the login modules, for example, using insecure hash functions with a weak collision probability.

Attacker's Limitations. Similar to [18], we assume that only few developers act maliciously, and they cannot compromise the source code review and software testing process. Moreover, they cannot compromise the deployed systems where the login module runs, so the operating system modules are trusted, which provide file reads, network communication, and random number generation. In other words, we do not consider the attacks in which the user-space login module with backdoors can directly control the kernel of operating system (e.g., a rootkit) or modify system libraries. In addition, we assume the backdoor creators cannot modify the password database in the deployed systems. Such attacks can be simply identified by database auditing.

B. Types of RCA Backdoor

A general RCA decision function $L()$, shown in Figure 1, takes inputs from users $\mathbb{U}$ and global states $\mathbb{G}$, maintains internal states $\mathbb{I}$, and returns a boolean value, $TRUE$ for accept and $FALSE$ for reject. Specifically, the response-computation component of $L()$ generates an expected response $Response'$, and the response-comparison component of $L()$ compares $Response'$ with the user's response $Response$. Based on how $Response$ is used in the login module with a backdoor, we classify backdoors into two categories:

Type T1: Bypassing response comparison *($Response' \neq Response$):* In this category, the attacker can authenticate successfully regardless of the value of $Response$ and $Response'$, because the response-comparison step is bypassed. The attacker can login even when the response-computation function $f()$ does not compute any response at all. The bypassing is usually triggered when some special conditions are met: Based on $L()$'s input type, there are three basic types of trigger conditions, listed as follows.

- $\mathbb{U}$-triggered backdoors. Special user inputs $\mathbb{U}$ can be used to trigger hidden logic or intended vulnerabilities in $L()$ to bypass the comparison statement

```
static int i=0;
...
i++ //record the login attempt number
if (i%10==0)
{
response = str.revert(challenge);
// transform based on the challenge
}
return response;
...
```

Figure 2. An internal-state triggered backdoor example.

and force $L()$ to directly return $TRUE$.

- $\mathbb{G}$-triggered backdoor. Global states $\mathbb{G}$, such as system times and MAC addresses, can also be used. For example, between 4:00am and 4:01am, $L()$ directly returns $TRUE$ regardless of user's response.
- $\mathbb{I}$-triggered backdoor. Internal states $\mathbb{I}$ can be a kind of trigger. For example, $L()$ can record the login failure frequency and return $TRUE$ if the failure frequency falls into a specified range.

Note that all these three kinds of trigger conditions and their combinations can be used to implement backdoors. We can eliminate this type of backdoors if we ensure that the response comparison cannot be bypassed.

Type T2: Controlling computation of expected response $(Response' = Response)$*:* In this category, the response comparison is not bypassed, but the response-computation component is affected by attackers. Recall that the response-computation function $f()$ generates a response $Response'$, which is compared to $Response$ to decide whether the user (or attacker) can login. However, as the developer who writes $f()$, the attacker can plant backdoors to make the response generated by $f()$ predictable. Ideally, the expected response should only depend on the challenge and the user's password. Based on how the backdoor in the response-computation function affects the expected response, we further classify this type of RCA backdoors into two sub-categories.

- ***Type T2a****: Response computation depends on information other than challenge and password.*

  In this category, when given a specific pair of password *pw* and challenge *cha*, the response-computation function can generates different responses, depending on the value of ($\mathbb{U}$, $\mathbb{G}$, $\mathbb{I}$). The attacker can predicate the response if when a special condition is planted in the login module.

  For example, the attacker can plant a hard-coded pair of username and password in $f()$. When the hard-coded username is supplied in $\mathbb{U}$, $f()$ uses the hard-coded password to compute a response, which makes the response predictable by the attacker.

  Figure 2 shows another example. The response computing function f generates a response that is the bit-wise inverse of the challenge every ten login attempts (i.e. internal states). So the backdoor attacker can always send the bit-wise inverse of the challenge to the server and authenticate successfully with a probability of 1/10.

  This type of backdoor can be eliminated if the computation function $f()$ is assured to depend only on cha and pw, i.e., for a given pw and cha, $f()$ always generates the same response. Therefore, the response-computation function $f()$ should only perform pure computation, without side-effect and dependencies to external or persistent internal states. This requirement can be satisfied if we ensure that the response-computation function is a *pure function*, which is introduced in Section II-C.

- ***Type T2b****: Response computation has collision-based backdoors.*

  In this category, when given specific password pw and challenge cha, the response-computation function $f()$ generates the same response $Response'$. $Response'$ is determined by the password pw, which is set by the normal user and is unknown to the attacker. Therefore, the attacker cannot predicate the exact response generated by $f()$.

  Given a challenge, if the output of $f()$ is different for different passwords (i.e. $f()$'s value space is uniform distributed), the attacker can only login successfully with a probability equals to that of guessing the right password. However, the malicious developer can implement a function with a high weak-collision probability. For example, given a challenge cha_0, $f()$ may output res_0 for half of possible passwords, which allows the attacker to login successfully with a probability of 1/2 using the response res_0.

  This kind of backdoor, called collision-based backdoor, can hardly be detected by traditional program analysis methods, but it can be eliminated if we can measure the computation function's collision probability. Again, it requires that $f()$ is assured to depend only on cha and pw.

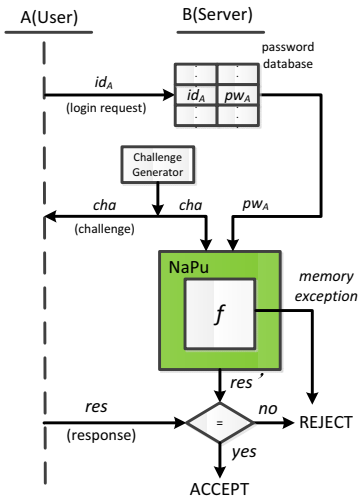

Figure 3. New RCA framework.

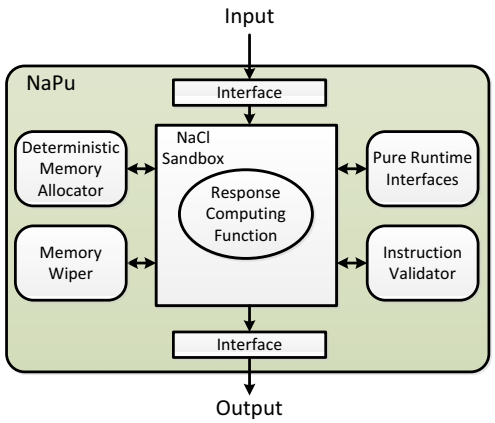

Figure 4. NaPu architecture.

From the above discussion, we can see that ensuring the response-computation function to perform pure computation is the basic to prevent Type T2 backdoors. Next, we introduce the concept of pure function.

C. Pure function

A function is called pure, if the following two properties hold [14]:

- It has no side effects, i.e., the evaluation does not have any visible effects other than generating the function results. Functions those modify arguments, global states or hard disk file are not free of side effects.
- Its execution is deterministic, i.e., given the same set of function arguments, it always generates the same result.

Pure functions introduce important features to simplify security analysis. For example, the password encrypt function $crypt()$ in POSIX should be a pure function. Given a plain password and a salt, $cyrpt()$ should return a fixed encrypted password. Moreover, $cyrpt()$ should be side-effect free, e.g., it should never modify any files on the disk or modify other global variables in the caller procedure's address space.

There are many solutions to provide pure function assurance. For example, Finifter et al. [14] presents a technique to implement verifiable pure functions in Java.

III. System Design

In this section, we introduce our new RCA framework, which uses a function-purifying sandbox to prevent backdoors. We introduce the framework in general, describe its key component — the NaPu sandbox, and illustrate the steps to eliminate backdoor from RCA login modules.

A. New RCA framework

A backdoor takes effect when it affects the decision of an authentication module. Strictly speaking, the authentication decision of an RCA module should only be based on a user's credentials on the server and the user's response to challenges. Other requirements, such as the valid time period to login, are not part of the core authentication process. However, when a system treats the authentication module as a blackbox, it is impossible to know how the decision is made.

To address this problem, we design an approach that partitions the RCA into components and ensures that the computation in each component is not affected by backdoors: Specifically, some components are trusted, such as the interfaces provided by the operating system; simple components can be verified by code analysis; other components are put into a sandbox environment to guarantee they are not affected by backdoors.

Figure 3 shows our new RCA framework. It first makes the response comparison explicit, so that it can be verified as a simple component. It also identifies other simple components that can be verified, such as lookup of password databases. Other components, including the response-computation function f is purified by the NaPu sandbox.

Based on this framework, we make formal analysis of the upper bound of backdoor usability (Section IV-B) and use a collision testing to estimate the maximum probability of a backdoor. If the probability is low enough, the backdoor cannot be practically used by attackers; Otherwise, our approach detects the backdoor.

B. NaPu: A native sandbox for purifying functions

The key component of our approach is a function-purifying sandbox called NaPu, a function-level sandbox environment for native binaries. It consists of an

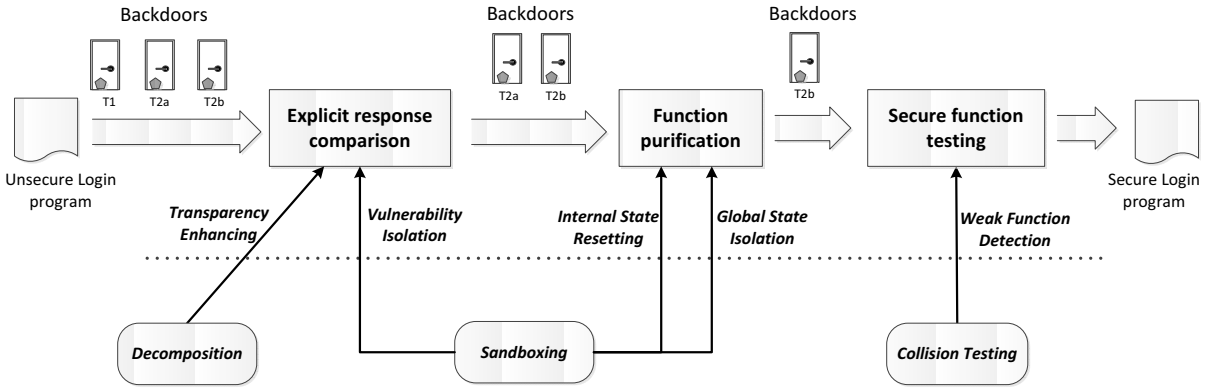

Figure 5. Countermeasures of defeating backdoors in RCA.

inner sandbox and four components: deterministic memory allocator, memory wiper, pure function interfaces and instruction validator, shown in Figure 4. We use the Native Client Sandbox [37] to provide a vulnerability isolation environment. The instruction validator of NaPu further makes sure that only deterministic instructions are allowed. For example, `CPUID` and `RDTSC` are prohibited. We use additional components to provide two additional properties.

Global state isolation. The pure function interfaces and instruction validator prevent the untrusted module from getting the global states that can be used as a trigger, such as the system time. If the function running in NaPu calls the special APIs or instructions to get global states, NaPu will throw an exception.

Internal state reset. The deterministic memory allocator and memory wiper will initialize the program states before every invocation of the untrusted module, to prevent backdoor from being triggered by persistent internal states. For each login request, if the allocated memory address is different, this address could be used to break the pure function requirement. Therefore we use the deterministic memory allocator to make sure that, for each login request, NaPu will start a new segment from a fixed address, and for each function executed in NaPu, its memory address is a fixed value. We also using the memory wiper to ensure all new allocated memory buffers are filled with zero to eliminate the uncertainty in memory allocation.

With these properties, the only two parameters that the NaPu receives are *challenge*, cha, generated by server and *password*, pw, from password database. Then we can confirm the response-computation function f is a pure function, which means the internal behavior of f is always the same no matter what input is. For the same cha and pw pairs, the computed response must be identical.

Such a sandbox can prevent a malicious developer from triggering backdoors using controllable inputs. It also makes sure the return value of f is only affected by cha and pw, which are not controllable by the malicious developer. So the backdoor tester and the malicious developer are in the same position and we can use collision testing to get the maximum probability of the backdoor.

C. Steps to eliminate backdoors using our RCA framework

Based on the framework and the NaPu sandbox, Figure 5 shows the overall process of securing RCA modules against backdoors, which consists of three steps:

1) Explicit response comparison: Explicit response comparison is used to make sure that the return value of L() is derived from comparison of expected response and the actual user response.

We divide the verification process L() into two steps: *response computation* and *response comparison* — to increase the execution transparency. Note that the response-computation function f usually contains a lot of cryptographic computations and is backdoor-prone. We put it in NaPu to prevent control flow hijacking (e.g. through exploiting vulnerabilities in f). This step can guarantee that the comparison statement cannot be bypassed, and eliminates T1-type backdoors directly.

2) Function purification: Once the response-computation function f is put in NaPu, and it can only take the explicit inputs (i.e. the challenge and password) as its arguments. Furthermore, for each run of the response calculating function f, this framework resets the memory used by the function and allocates memory in a fixed manner. In this way, it prevents backdoors from taking advantage of global or internal states. As a result, the computation function is pure and thus T2a-type backdoors are eliminated.

3) Backdoor usability testing: For T2b-type backdoors, we use collision testing to verify whether the response computing function f is consistent and has no special characteristics such as high collision probability. According to theorem and corollary in Section IV-B, our testing method is straightforward. We compute the upper bound of a backdoor usability. If the upper bound is lower than a practical threshold, we assert that there is no usable backdoor in the login module.

As long as the f can only receive challenge cha and password pw, both of which are random to malicious developers or testers, we randomly generate plenty of cha, and for each cha, randomly generate plenty of pw, to check the weak collision probability of the response computing function. For different cha or pw, the response weak collision probability should be small enough to prevent a malicious developer from authenticating to the system. If the probability is abnormally high, we detect the backdoor. For example, if the login program has the backdoor shown in Figure 2, when the challenge is fixed, we will get 1/10 of the response are the inversed challenge, although we feed different passwords.

IV. Security Analysis

In this section, we define response-computable authentication (RCA) and usable RCA backdoors. We then analyze the existence and usability of backdoors in our framework. Finally, we discuss the countermeasure against possible backdoors.

A. RCA backdoors definition

Definition 1 (**Response-Computable Authentication**)**.** A two-party (e.g. client and server) authentication is called a response-computable authentication (RCA) if all the following statements hold:

- The client computes (or chooses) a response and then sends to the server.
- The server has a computation component that computes an expected response independently.
- Whether the authentication passes or not depends on whether the user response equals to the expected response.

In brief, an RCA is an authentication with a *compute-then-compare* logic. Specifically, the server and the client compute the response separately. The server compares the two values to decide whether to authenticate the user.

The server's computation component of an RCA can be modeled as a computation function f. We present the details in the following definition.

Definition 2 (**Computation Function**)**.** A computation function is a binary function, which reads in two arguments that cannot be controlled by attackers, i.e., a randomly generated challenge cha and a secret password pw from the password database, and then outputs an expected response $Response'$ for further comparison.

In our framework, the computation component resides in the sandbox NaPu, and thus the computation function f is a *pure function*, i.e. given the same arguments *cha* and *pw*, f returns the same result and does not cause any side effects.

Definition 3 (**Usable RCA Backdoor**)**.** Given an RCA whose computation function is f, a RCA backdoor exists in this RCA if and only if, there is a hidden client-side response generation schema S such that the attacker can login successfully with a probability $P_{backdoor}$. This backdoor is called a $(S, P_{backdoor})$-backdoor. If $P_{backdoor}$ is higher than a predefined threshold $P_{threshold}$, it is called *usable*.

More specifically, the attacker can choose a special (or any) id and waits for a special (or any) cha, and then sends the response $S(id, cha)$ to server, and finally logins successfully with a probability

$$P_{backdoor} = P\left(S(id, cha) == f(pw, cha)\right),$$

such that,

$$P_{backdoor} \geq P_{threshold},$$

where the pw is the corresponding secret password of id.

The attacker is also the developer of the computation function f, and he/she has some knowledge about f, such as f's algorithm or a hidden vulnerability in f's implementation or a rarely executed execution path. These knowledge can be combined into the attack schema S. As a result, even though the attacker does not know id's corresponding pw, he/she can login successfully with a probability (i.e. $P_{backdoor}$) greater than the probability (denoted as P^{id}_{guess}) of guessing the right password. Even worse, if the attacker can try several login attempts in a short time, he/she can login successfully much faster.

Deduction 1 (**Probability of Successful Attacks in a Login Session**)**.** In a classical login session, the user is allowed to try several login attempts before he/she is forbidden to login. Suppose the count of permitted attempts in a login session is N_L and the attacker can login successfully with a probability $P_{backdoor}$ in each attempt, the probability with which the attacker can login successfully in the whole login session is

$$P_{total} = 1 - (1 - P_{backdoor})^{N_L}$$

According to the Bernoulli inequality, we can infer that: $P_{\text{total}} \leq P_{\text{backdoor}} \times N_L$. Especially, when $P_{\text{backdoor}} \ll 1$, the statement $P_{\text{total}} \approx P_{\text{backdoor}} \times N_L$ holds. As a result, if $P_{backdoor}$ and N_L are big enough, the attacker can easily login into the target system. On the other hand, a practical secure login system should make sure that $P_{backdoor}$ in each attempt is small enough and the count of permitted attempts in a certain time period is small.

B. Security analysis of our framework

1) Effectiveness on different types of backdoors:

(1) Type T1 backdoors: For T1 type backdoors, the compute-then-compare logic is violated, e.g., through vulnerability exploiting. In our framework, the login module is decomposed into several components, e.g., the computation and comparison components. Besides, the computation component is protected by the sandbox NaPu which is robust against vulnerabilities, and thus the control flow cannot be controlled by attacker. Meanwhile, the comparison component is quite simple (i.e. a compare statement) and can be enforced easily. With this explicit comparison enforcing mechanism, the compute-then-compare logic cannot be violated if the control flow reaches the computation module. Moreover, other control-flow-integrity techniques can be applied to ensure the computation module not be bypassed. As a result, there are no T1 type backdoors in our framework.

(2) Type T2a backdoors For T2a type backdoors, given a certain input pw and cha, the output $f(pw, cha)$ is not deterministic, e.g. f outputs a predefined response when a specific trigger condition (e.g. a specific date) meets and outputs a normal response otherwise. In our framework, f resides in a sandbox NaPu, and we use global state isolation and internal state resetting to make sure f is a pure function and thus deterministic. So, there are no T2a type backdoors.

(3) Type T2b backdoors. However, in our framework, even though the compute-then-compare logic is enforced and the computation function is deterministic, backdoors of type T2b can still be active.

In this framework, attackers only know id, cha and f's algorithm. Besides, f is a pure function, and its result is used in an enforced comparison to make a decision. In order to login successfully, attackers have to provide a valid response matches the deterministic expected response. More specifically, they must choose a special (or any) id_0 and wait for a special (or any) cha_0, and then send a response $S(id_0, cha_0)$ to the server to match the expected response $f(pw, cha_0)$ generated by the server.

Fortunately, attackers know nothing about pw although pw is determined by id chosen by attackers. Ideally, if the outputs of $f(pw, cha_0)$ are completely different for all pws (i.e. f's value space is uniform distributed), the attacker-supplied response $S(id_0, cha_0)$ matches $f(pw, cha_0)$ only with a low probability of $1/M$ (let M be the count of possible passwords), i.e. attackers can login successfully with a probability of 1/M which equals to the negligible probability of guessing id's password.

However, the outputs of $f(pw, cha_0)$ may be same for some different pws. In an extreme case, all outputs are the same for all different pws. And then the attacker can choose any password, and behaves like a legal user to generate a response, and logins successfully. In other words, there may be backdoors if f's value space is not uniform distributed.

Unfortunately, the attacker knows f's algorithm, and thus knows the image (i.e. the value space) of f. As a result, $S(id_0, cha_0)$ may match $f(pw, cha_0)$ with a high probability and thus an usable backdoor exists. For example, for a certain cha_0, if half of the pws lead to a same output $f(pw, cha_0)$ (denoted as res_0, known by attackers), the attacker can randomly choose an id and waits for the special cha_0 sent from the server, then he/she sends res_0 back to the server, and finally logins successfully with a probability equals to $1/2$. Obviously, this is an usable backdoor.

But these backdoors' usability can be measured and controlled as following, so they can hardly be used practically.

2) Usability of backdoors in our framework:

Premise 1. Before discussing the usability of backdoors in our framework, we declared some premises which are enforced by the framework or usually acceptable.

- The attacker knows every possible valid id.
- For any id, the attacker knows nothing about its corresponding pw, i.e. the pw is random.
- The attacker knows f's algorithm.
- The computation function f is pure, and thus its output is deterministic when given inputs pw and cha.
- The computation-then-compare logic cannot be violated.
- From the cha, figuring out the pw is impossible (i.e. they are independent).

Definition 4 (Collision probability of a computation function). Given a pure response computation function $f(pw, cha)$, suppose there are M possible passwords.

- For a certain cha_0, let $\{res_1, res_2, ..., res_k\}$ be the image of $f(pw, cha_0)$. Besides, there are M_i passwords which may cause $f(pw, cha_0) = res_i$, where $i = 1, 2, ..., k$ and $M_1 + M_2 + ... + M_k = M$. Then we define $P_{col}^{cha_0} = max\{M_1, M_2, ..., M_k\}/M$.

- For all possible cha_0, we denote the maximum $P_{col}^{cha_0}$ as P_{col}^{max}. This probability is called the $collision\ probability$ of the computation function f.

Theorem 1 (**Usability of backdoors in our framework**). *Given any RCA implemented in our framework, suppose its computation function is f whose collision probability is P_{col}^{max}, then any T2b-backdoor attackers can only login successfully with a probability not greater than the collision probability, i.e. $P_{backdoor} \leq P_{col}^{max}$.*

The proof to Theorem 1 is listed in the appendix A.

C. Countermeasures against T2b backdoors

As discussed earlier, only T2b type backdoors may exist in our framework. More specifically, if and only if the computation function f's value space is not uniformly distributed, backdoors exist in our framework. According to the Theorem 1, T2b backdoor attackers can only login successfully in our framework with a maximum probability of P_{col}^{max}. For any computation function f, if the corresponding P_{col}^{max} is assured to be smaller than the threshold $P_{threshold}$, then we can conclude there is no usable backdoors in our framework.

In our framework, a collision testing is made to evaluate the computation function's collision probability P_{col}^{max}. If this probability is too high, i.e. higher than the threshold, a backdoor alert is triggered.

However, figuring out the exact collision probability of a computation function is computationally hard. According to the Definition 4, in order to compute the collision probability, P_{col}^{cha} should be computed for all cha. Further, for a given cha_0, all possible pw should be traversed to compute $P_{col}^{cha_0}$. Notice that the counts of cha and pw are both huge, and thus it is hard to compute the exact collision probability of a computation function.

Alternately, we sample parts of chas and parts of pws and then compute a similar collision probability to emulate the exact collision probability. More specifically, some random challenges $\{cha_1, cha_2, ..., cha_c\}$ are chosen first. And then for any chosen cha_i, some random pws are selected to test f, and thus $P_{col}^{cha_i}$ is computed. Then the collision probability is computed, i.e. $P_{col}^{max} = max\{P_{col}^{cha_1}, P_{col}^{cha_2}, ..., P_{col}^{cha_c}\}$. If the sampled collision probability is larger than the backdoor threshold, the computation function is regarded vulnerable and should be eliminated before being deployed.

Although this sampling and testing scheme cannot deduce the exact collision probability of a computation function, it can exactly model attackers' capabilities. In other words, if attackers find a usable backdoor in the computation function, security testers in our framework is likely to get a P_{col}^{max} higher than the threshold.

Notice that passwords in the real database only cover a small part of all possible passwords. Besides, passwords are random. And thus, even though the attacker knows f's algorithm, he/she does not know the exact distribution of f's value space for passwords in the real database. For example, if the outputs of f are same (denoted as res_0) for half of all possible passwords, but all passwords in the database are not in this half, then the attacker can never login successfully if he/she still chooses res_0 as the response. So, attackers can not predicate which response are more likely to be correct because the password database is small and random.

On the other hand, the security testers in our framework randomly choose some passwords to evaluate P_{col}^{max}. This randomly sampling can model the real password databases.

As a result, the attackers have no advantage over the testers. And thus, with the help of our thorough and random collision testing, computation functions with a high collision probability can be filtered out or no usable backdoors exist.

V. Implementation

In this section, we present the implementation of our approach, with focuses on NaPu — the Native Pure-function enforcing sandbox. We build NaPu based on Google's Native Client (NaCl) [37]. NaPu provides vulnerability isolation inherited from NaCl.

Vulnerability isolation. NaCl consists of two layers of sandbox. The inner sandbox makes a secure subdomain in the native process. Besides applying software isolation [37] and disabling the unsafe machine instructions, it also enforces structural rules and alignment to make disassembling reliable, and constrains references in both instruction and data memory by segmenting. The outer sandbox uses secure interfaces to control interactions between modules and the system.

Even though there were vulnerabilities in the code in the sandbox, the effect will be contained in the sandbox and cannot be used to exploit the system.

Pure function enforcing. To ensure that the response computing function f running in NaPu is pure, we need to eliminate the non-deterministic source of f by isolating the global states, resetting the internal states, and making sure it has no side effects. Meanwhile, all the instructions allowed in NaPu are deterministic.

Global state isolation. NaCl provides a white-list mechanism to confine the native code's accesses to system calls. Those trusted system calls include file operations, timers, socket calls, threading operations, debug mode calls and sound/graphic interfaces. Those system calls can access global states and use them as

the trigger to generate a special response, which are not necessary for normal response computing. Therefore, they are all prohibited in NaPu, except the system calls related to memory allocation. Specifically, NaPu only provides pure function interfaces and disallows the native code to invoke the following system calls:

- The system calls that can perform file operations (e.g., `open` , `read`, `write`).
- The system calls that can obtain system timers (e.g., `gettimeofday`, `clock`, `nanosleep`).
- The system calls that can make network accesses (e.g., `accept`, `connect`, `recvmsg`).
- The system calls that can obtain system information (e.g., `sysconf`, `getpid`).
- Others (threading related, sound/graphic related, debug related).

In addition, NaPu disables the following x86 instructions in the native code of f through the instruction validator, so that NaPu makes sure that all the allowed instructions are deterministic.

- Instructions that can obtain hardware identification configuration (e.g., `CPUID`) ;
- Instructions that can obtain hardware performance information (e.g., `RDPMC`, `RDTSC`, and `RDTSCP`);
- Instructions that can access I/O ports (e.g., `IN/OUT`).

Internal state reset. The backdoor may use a local variable as the trigger, such as the example shown in Figure 2. NaPu uses the memory wiper to resets the internal states each time to call f, to make sure there is no variable recording internal states.

Even though there are only memory allocation system calls left in NaPu, the allocated memory or the memory address can be used as triggers when there is no special handling. For instance, an attacker may initiate a number of login requests in a short time, since each login authentication will be performed in one sandbox, the memory address for each sandbox will be different for each login request. Thus, the attacker can break the pure function requirement. To deal with such situations, NaPu uses the deterministic memory allocator. The allocator fills all new allocated memory buffers with zeros to eliminate the uncertainty in allocated memory blocks. Further, for each request, NaPu will start a new segment start from address 0x10000, and for each function executed in NaPu, its memory address is a fixed value. Finally, any instruction that can get the value of segment registers is disallowed.

Memory exception handling. Memory allocation may fail. To deal with this situation, one solution is to suspend the execution until free memory appears. But this method is easy to create a deadlock, hence it is not suitable for most pure computing scenarios. Another solution is to return an error. However, if this error information is returned directly to the inner function in NaPu, the inner function can generate different values according to the allocation results, which violates the pure function characteristics. NaPu takes a simple method that the outermost function in NaPu will be stopped as failure and its caller will get an exception if any memory allocation fails. The framework will catch this exception and directly reject the login attempt.

Mechanism	Response computing function f
PLAIN	Base64(pw)
CRAM-MD5	Base64($HMAC_{\text{MD5}}(pw, nonce)$)
HOTP	Truncate($HMAC_{\text{SHA-1}}(pw, C)$)
TOTP	Truncate($HMAC_{\text{SHA-1}}(pw, time)$)

Table I
RESPONSE-COMPUTATION FUNCTION f OF DIFFERENT AUTHENTICATION MECHANISMS.

Through global state isolation and internal state reset, the only inputs of response-computation function are the challenge and the password. Meanwhile, all the instructions allowed by NaPu are all deterministic. Then for the same cha and pw pairs, the return values of f will be identical, which meets the requirements of the theorems in Section IV-B.

NaPu and DeterministicExecution [1] are parallel projects that are independent to each other. The DeterministicExecution is a subproject of NaCl to disabling sources of non-determinism for guest code, while NaPu's scenario is more succinct because response computing function usually only takes arithmetic operations.

VI. EVALUATION

In this section, we describe our results in porting several widely used RCA modules into our framework, measuring their performance overhead in NaPu, and checking whether they are backdoor-free. We also did experiments to show the ability to detect real-world backdoors. In our experiments, NaPu is deployed in the authentication module, and the authentication is performed on the server side. The performance evaluation was carried out on a Linux Ubuntu server 10.10 with Intel Core2 Duo CPU at 2.40GHz. Because the response computing function f is pure, we can test its property in parallel when needed.

We evaluated NaPu by porting response computing functions of different RCA mechanisms into our framework. These RCA mechanisms are from widely adopted authentication layer SASL and OTP, shown in table I.

- In *PLAIN*, the challenge is a simple query for login, and the response is the password encoded by base64. Hence f = Base64(pw).
- In *CRAM-MD5*, the challenge is a base64 encoded $nonce$. When receiving the challenge, the user de-

Mechanism	pw length(bit)	Original Time(μs)	NaPu Time(μs)	Runtime Overhead
PLAIN	128	0.0923	0.0917	-0.65%
CRAM-MD5	128	1.8121	1.8588	2.57%
HOTP	128	3.5615	4.1756	17.24%
TOTP	128	3.6286	4.1652	14.78%
PLAIN	256	0.1704	0.1555	-8.74%
CRAM-MD5	256	1.8962	1.9342	2.00%
HOTP	256	3.6748	4.2393	15.36%
TOTP	256	3.6512	4.2267	15.76%

Table II
RUNTIME OVERHEAD OF NAPU.

codes it to get the $nonce$, then hash it using HMAC with the user's password pw as the secret key. Therefore $f = Base64(HMAC(pw, nonce))$.

- In *HOTP*, the challenge is an increasing counter, usually 8-byte. We denote it as C. The response is a truncated value on HMAC of the key pw and the counter C. Therefore f=Truncate($HMAC_{\text{SHA-1}}(pw, C)$). Here *Truncate* represents the function that converts an HMAC-SHA-1 value into an HOTP value which usually a 6-digit number.
- In *TOTP*, the challenge is an anticipated time, the response is a truncated value on HMAC of the key pw and the time-stamp $time$. Therefore f=Truncate($HMAC_{\text{SHA-1}}(pw, time)$). Here *Truncate* represents the function that converts an HMAC-SHA-1 value into an TOTP value which usually a 6-digit number.

A. Performance overhead

We compared the performance of these response-computation functions ported in NaPu with its original implementation. For each function, we measured 10^7 times for both 128-bit pw and 256bit pw and then averaged the time measurements. These functions seldom use heavy memory operations, so NaPu only introduces initial reset and more API/OPCode restrictions over NaCl, and the performance overhead is about the same as NaCl.

The results, shown in Table II, show that the maximum performance overhead is less than 20%, and the single processing time is less than 5 μs, which is acceptable in real login modules. The interesting part is in PLAIN mechanism, the f in NaPu is faster than native code. The reason is that NaPu uses 32-bytes block alignment which is more suitable for Base64 transformation.

B. Backdoor usability testing

1) Ported login modules: We conducted experiments to measure the weak collision probability of response computing functions in these ported login modules.

We set the backdoor usability threshold P_{thres} as 0.01%, which means that an attacker can get no more than 1 successful login in 10,000 attempts.

We had different test cases for different scenarios of cha. We designed these experiments as the following:

- For PLAIN, the challenge is not involved in the response computing. We generated 10^5 random passwords as the input of f, and got 10^5 responded values. The password was 128 bit and the response was 144 bit. The weak collision probability (i.e., the upper bound of $P_{backdoor}$) was almost 0.
- For CRAM-MD5, the challenge is random. We randomly generate 10^6 cha. For each cha, we generated 10^5 random pw to test its P_{col}, and then got their maximum number as $P_{\text{col}}^{\text{max}}$.
- For TOTP, the challenge is predictable. We assumed cha is increased every 30 seconds and we generated all possible cha for the next ten years, which are 10 million different cha; For each cha, we randomly generated 10^5 pw to test its P_{col}, and then got their maximum number as $P_{\text{col}}^{\text{max}}$.
- For HOTP, the challenge is predictable. We assumed the counter is increased 100 times at most every day, which means that a person normally logins less than 100 times per day. We generated 365000 cha for the next 10 years. For each cha, we randomly generated 10^5 pw to test its P_{col}, and then get their maximum number as $P_{\text{col}}^{\text{max}}$.

All passwords used in experiments are 128-bit long. The output response length is from RFC standard. All of these tests are completed within a week using 10 severs in parallel. The results, shown in Table III, present that all $P_{\text{col}}^{\text{max}}$ are less than 10^{-4}, so P_{backdoor} is less than $P_{\text{thres}} = 0.01\%$. So we can assert that there are no usable backdoors in these authentication mechanisms.

2) Volunteer-created backdoor: To test the framework's ability to detect the backdoor, we asked some student volunteers who study computer science to create different backdoors. Because most of the existing login backdoors are using pre-set special credential to circumvent the authentication, which is naturally eliminated in our framework, for simplicity, we limited the scope of Type-2 backdoors so that the response should be compared explicitly. The students can use different trigger conditions or modify hash functions but the challenges were randomly generated.

We used our testing strategy to test their backdoor probability. The framework caught all of these backdoors. There were two representative backdoors in these experiments. Both of them can reach a very high $P_{\text{col}}^{\text{max}}$. One is 100% and the other is 67%.

We asked the backdoor creators to explain their methods. One used a special challenge as the trigger. He first changed the challenge to bit form. If the fourth bit

Mechanism	pw len(bit)	cha len(bit)	res len(bit)	cha num ($*10^3$)	pw num per cha	$P_{\text{col}}^{\max}$ ($/10^5$)	Time (hours)
PLAIN	128	None	144	None	10^5	0	≈ 0
TOTP	128	64	20	10000	10^5	2	1065
HOTP	128	64	20	365	10^5	2	30
CRAM-MD5	128	64	128	1000	10^5	0	97

Table III
COLLISION TESTING OF PORTED RESPONSE COMPUTING FUNCTION f.

of the challenge is 1 and the fifth bit is 0, then f outputs a special response. Then he tried several times and waited for an appropriate challenge to log in. Basically he had 1/4 probability to log in successfully. During our testing, we easily found this backdoor because the $P_{\text{col}}^{\max}$ is 100%.

In another case, the hash function was wrapped to be insecure. The creator first computed the response as normal. Then he mapped those response value to a small space with high collision probability to some special responses, which means $f()$ compresses the image of a secure hash function. Through the standard testing, we discovered this malicious response computing function with $P_{\text{col}}^{\max}$ = 67%.

VII. DISCUSSION AND LIMITATION

Since either malicious developers or outsourced contractors have chances to instrument backdoors, backdoors are a constant threat for large software systems [19]. Advanced attackers can even plant backdoors into cryptography algorithms which are even more difficult to discover. For reference, even for well-known algorithms such as DES and MD5, their weaknesses were found after they had been used widely for many years.

Instead of trying to detect all possible backdoors in software, this paper focuses on preventing usable backdoors in response-computable login modules, which was widely used in many authentication systems.

In our login framework, we need several trusted modules (see Section IV). In fact, these trusted modules contain very simple logic, such as file read and socket communications. In general, operating systems directly implement these functionalities. In addition, we also need a trusted random number generator, which is also a basic kind of system service (such as `/dev/urandom` in Linux systems).

Thus, unless the backdoor creator can take control of operating systems, we think existing mechanisms such as manual code review and program verification approaches can ensure that the assistant modules are trusted. Furthermore, techniques such as CFI/SFI can be used to prevent an attacker from bypassing the authentication procedure. On the other hand, for the enterprises which publish user-space software, malicious programmers hardly compromise the OS kernel. How to protect OS kernel is out of the scope of this paper.

In real-world systems there may be other requirements. For example, a user is only allowed to login between 8am and 6pm or only from a specific set of IP addresses. The system can enforce these constraints before reading the password from the database outside NaPu, so that the pure function in NaPu only handles password and challenge.

This paper assumes a uniform probability distribution for the testing selection space for the password. In the future, approaches of more adaptive distributions [10] can be used.

A possible attack is timing-based attack [7] that used to guess passwords. However, from the table II we can see that the time taken by computation functions is so little that its variation can hardly be measured externally. If necessary, intentional delays can be inserted to prevent this attack. In our framework the attacker cannot have any other information to launch a side-channel attack.

VIII. RELATED WORK

Backdoor detection based on network traffic behavior. To detect backdoors which can be triggered remotely, many studies focus on analyzing network traffics based on network intrusion detection systems (NIDS). Zhang and Paxson [39] developed a set of algorithms that exploit many novel characteristics such as the frequency of small packets, the size of packets, connection directionality and keystroke times to detect machine-driven interactive backdoors. Based on the work [39], Gonzalez et al. [17] proposed traffic sampling and filtering methods and further implemented a backdoor detector. Besides traffic characteristics, Horng et al. [20] adopted the Dynamic Link Library (DLL) injection technique to record all DLLs used by the target application, and took advantage of these extra characteristics to determine whether the target application has backdoors.

These methods prevent certain type of backdoors that can cause anomalous network behaviors. However, these methods can be evaded if backdoors do not cause anomalous network behaviors. For example, during a normal login connection, an attacker directly uses a

backdoor username or super password to log in to a system.

Backdoor detection by program analysis. Naturally, many malware detection or analysis methods can be also applied to backdoor detection. For example, to detect hidden malware time-bombs, Crandall et al. [13] implemented a temporal search technique that runs virtual machines at slightly different rates of time.

To explore malware code space, Wilhelm et al. [36] used a forced-execution approach to force a driver to run different paths. Moser et al. [23] and Brumley et al. [9] independently implemented similar symbolic-execution-based malware analysis systems to explore different paths in malware. Comparetti et al. [12] proposed a graph-match-based solution to determine the malicious functionalities in malware samples.

These techniques can be used to detect certain types of backdoors. However, there are also many anti-analysis techniques, such as code obfuscation [11], [22], [25] and code encryption [28], [38]. Instead of detecting backdoors, we focus on nullifying the effects of a backdoor in RCA login frameworks.

Backdoor prevention. Wysopal [29] gave an overview on different backdoor mechanisms and malicious indicators, and suggested using static analysis to identify backdoor indicators such as static variables that look like hashes or cryptographic keys, to prevent the pre-owned special credentials.

Source code review cannot detect the backdoors that are instrumented at the binary level. Wheeler [35] proposed a diverse double-compiling technique that compares the untrusted binary with the one generated by another trusted compiler. In addition, syntax-based [15] or semantics-based [16] binary comparison techniques can also be used to statically identify the equivalence of binaries. The deviation detection method in [8] can dynamically identify the equivalence of execution traces. These studies could alleviate the backdoor problems caused by malicious compilers.

Hardware backdoor. In recent years, hardware backdoors became a hot topic [18], [31], [33], [34]. To defend against malicious hardware, Hicks et al. [18] proposed BlueChip, a hybrid hardware/software approach. BlueChip identifies unused circuits during the verification tests, and uses trusted software to emulate the unused circuits. Thus, even if the unused circuits contain backdoor logic, they cannot be activated. This method could be applied to isolate certain types of backdoors in software. For example, if some pieces of code in a login module are not tested, we can separate them from the login module. However, there are many flexible backdoor implementation methods. Simple replacing unused code in login modules cannot address the issue.

Waksman et al. [33] proposed a method to make the backdoor intractable to attackers by scrambling the inputs. Due to the input data randomization, backdoors are hardly controlled by attackers. Preventing the untrusted login modules from receiving expected inputs is very useful to defend backdoors. However, in login protocols, many data, such as the challenge values, responses, cannot be scrambled. Thus the method cannot be directly used in login protocols.

IX. Conclusions

Response-computable authentication (RCA) is a two party authentication model widely adopted by many login systems. Unfortunately, these systems have suffered from the threats of backdoors. A malicious developer could leave backdoors in source code, through malicious compilers, by planting delicate vulnerabilities, or even through weak cryptography algorithms. Traditional technologies have difficulty in completely eliminating backdoors from login systems.

In this paper, we propose a framework for RCA systems to ensure that no usable backdoor exists. And we prove theorems about the upper bound of potential backdoor usability. Our framework splits the RCA model into one response-computation function and some assistant logic. These assistant logic are fairly simple and can be checked manually. The response-computation function is usually complicated and may have backdoors. We put this function into NaPu, a Native pure-function-enforcing sandbox. NaPu can prevent an attacker from triggering backdoors via vulnerability and ensure the response computing function pure. We prove theorems to give the upper bound of backdoor usability in a login module, which forms a theoretical basis of our testing methods. Through the testing, either we can detect the possible backdoor or we can ensure the backdoor cannot be used by its creator. The idea of enforcing functional purity could be used for multiple applications beyond login modules, such as e-voting machines [14], [27], [30]. How to extend the pure function characteristic to enhance program security is our future work.

We ported several popular login modules into this framework and verified that they are backdoor-free. We also detected some real backdoors in login module via probability testing. Our performance measurement showed acceptable overhead. The result of automatic standard tests shows that the framework can be applied to real login systems to ensure no practically usable backdoor.

Acknowledgments

We are grateful to David Wagner, Shuo Chen, Prateek Saxena, and the anonymous reviewers for their insight-

ful comments and suggestions. This research was supported in part by National Natural Science Foundation of China (Grant No. 61003216), National University of Singapore under NUS Young Investigator Award R-252-000-378-101, and the Office of Naval Research under MURI Grant No. N000140911081.

REFERENCES

[1] NaCl project:Disabling sources of non-determinism for guest code. http://code.google.com/p/nativeclient/wiki/DeterministicExecution.

[2] ProFTPD Backdoor Unauthorized Access Vulnerability. http://www.securityfocus.com/bid/45150.

[3] RSA SecurID Two-factor Authentication. http://www.rsa.com/node.aspx?id=1156.

[4] TOTP: Time-based One-time Password Algorithm. http://tools.ietf.org/html/draft-mraihi-totp-timebased-08.

[5] Back Door in Commercial Shopping Cart. 2000. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2000-252.

[6] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE SYMPOSIUM ON RESEARCH IN SECURITY AND PRIVACY*, pages 72–84, 1992.

[7] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[8] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of USENIX Security Symposium*, Aug. 2007.

[9] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Book chapter in "Botnet Analysis and Defense", Editors Wenke Lee et. al.*, pages 65–88. Springer US, 2008.

[10] C. Castelluccia, M. Durmuth, and D. Perito. Adaptive password-strength meters from markov models. In *Proceeding of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, 2012.

[11] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998.

[12] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying Dormant Functionality in Malware Programs. In *2010 IEEE Symposium on Security and Privacy*, pages 61–76, 2010.

[13] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: detecting hidden malware timebombs with virtual machines. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 25–36, New York, NY, USA, 2006. ACM.

[14] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in java. *Proceedings of the 15th ACM conference on Computer and communications security CCS 08*, page 161, 2008.

[15] H. Flake. Structural comparison of executable objects. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2004.

[16] D. Gao, M. K. Reiter, and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proceedings of the International Conference on Information and Communications Security*, pages 238–255. Springer-Verlag, 2008.

[17] J. Gonzalez and V. Paxson. Enhancing Network Intrusion Detection with Integrated Sampling and Filtering. In *Recent Advances in Intrusion Detection (RAID)*, volume 4219 of *Lecture Notes in Computer Science*, pages 272–289. Springer Berlin / Heidelberg, 2006.

[18] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *2010 IEEE Symposium on Security and Privacy (SP)*, pages 159–172, 2010.

[19] L. Ho and A. Atkins. Security of software outsourcing in military and government agencies. In *Proceedings of IADIS International Conference on WWW/Internet 2005*, pages 347–355, 2005.

[20] S.-J. Horng, M.-Y. Su, and J.-G. Tsai. A dynamic backdoor detection system based on dynamic link libraries. *International Journal of Business and Systems Research*, 2(3):244–257, 2008.

[21] J. Klensin, R. Catoe, and P. Krumviede. IMAP/POP AUTHorize Extension for Simple Challenge/Response. RFC 2195, Internet Engineering Task Force, Sept. 1997.

[22] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[23] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *SP '07. IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[24] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. HOTP: An HMAC-Based One-Time Password Algorithm. RFC 4226, Internet Engineering Task Force, Dec. 2005.

[25] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Berkeley, CA, USA, 2007.

[26] A. Salomaa. *Public-Key Cryptography*. Springer, 1996. ISBN 3-540-61356-0.

[27] N. Sastry, T. Kohno, and D. Wagner. Designing voting machines for verification. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[28] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[29] T. Shields and C. Wysopal. Detecting Certified Preowned Software. In *BlackHat-Europe*, 2009.

[30] C. Sturton, S. Jha, S. A. Seshia, and D. Wagner. On voting machine design for verification and testability. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 463–476, New York, NY, USA, 2009. ACM.

[31] C. Sturton, D. Wagner, and S. T. King. Defeating UCI: Building Stealthy and Malicious Hardware. In *32th IEEE Symposium on Security and Privacy*, 2011.

[32] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[33] A. Waksman. Silencing Hardware Backdoors. In *32nd IEEE Symposium on Security and Privacy*, 2011.

[34] A. Waksman and S. Sethumadhavan. Tamper Evident Microprocessors. In *IEEE Symposium on Security and Privacy*, pages 173–188, 2010.

[35] D. A. Wheeler. Countering Trusting Trust through Diverse Double-Compiling. In *21st Annual Computer Security Applications Conference*, Tucson, Arizona, 2005.

[36] J. Wilhelm and T. cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, pages 219–235, 2007.

[37] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[38] A. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 129, Washington, DC, USA, 1996. IEEE Computer Society.

[39] Y. Zhang and V. Paxson. Detecting backdoors. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, page 12, Denver, Colorado, 2000. USENIX Association.

APPENDIX

Theorem 1 (Usability of backdoors in our framework) *Given any RCA implemented in our framework, suppose its computation function is $f()$ whose collision probability is P_{col}^{max}, then any T2b-backdoor attackers can only login successfully with a probability not greater than the collision probability, i.e. $P_{backdoor} \leq P_{col}^{max}$.*

Proof: Assume there is a backdoor $(S_0, P_{backdoor})$ in the RCA implemented in our framework such that $P_{backdoor} > P_{col}^{max}$, i.e. there is a client-side response generation schema S_0 such that the generated response matches the expected response with a probability higher than P_{col}^{max}.

More specifically, the backdoor attacker can choose a special (or any) id_0 and wait for a special (or any) cha_0, then sends back $S_0(id_0, cha_0)$ to the server for further comparison. Finally, the server-side comparison passes with a probability higher than P_{col}^{max}. In other words, the following statement holds:

$$\begin{aligned} P_{backdoor} &= P\left(S_0(id_0, cha_0) == f(pw, cha_0)\right) \\ &> P_{col}^{max} \end{aligned} \tag{1}$$

for a certain S_0, id_0 and cha_0.

On the other hand, for any response generation schema S' used by the attacker, for any id' and cha' chosen by this schema, the attacker can login successfully with a probability

$$P(S', id', cha') = P\left(S'(id', cha') == f(pw, cha')\right).$$

Let $\{res_1, res_2, ..., res_k\}$ be the image of $f(pw, cha')$, and M is the count of all possible pws. Besides, there are M_i passwords which may cause $f(pw, cha')$ equals to res_i, where $i = 1, 2, ..., k$ and $M_1 + M_2 + ... + M_k = M$. Then, there are two cases:

Case 1. The client-supplied response $S'(id', cha')$ does not exist in the set $\{res_1, res_2, ..., res_k\}$, i.e. there is no i such that $res_i = S'(id', cha')$. As a result,

$$\begin{aligned} P(S', id', cha') &= P\left(S'(id', cha') == f(pw, cha')\right) \\ &= 0 \qquad \leq P_{col}^{max} \end{aligned}$$

Case 2. The client-supplied response $S'(id', cha')$ exists in the set $\{res_1, res_2, ..., res_k\}$, i.e. there is an i such that $res_i = S'(id', cha')$. Besides, according to the Premise 1, pw is random. As a result,

$$\begin{aligned} P(S', id', cha') &= P\left(S'(id', cha') == f(pw, cha')\right) \\ &= P\left(res_i == f(pw, cha')\right) \\ &= \tfrac{M_i}{M} \qquad \leq \tfrac{\max\{M_1, M_2, ..., M_k\}}{M} \\ &= P_{col}^{cha'} \qquad \leq P_{col}^{max} \end{aligned}$$

In a word, the following statement holds:

$$\begin{aligned} P(S', id', cha') &= P\left(S'(id', cha') == f(pw, cha')\right) \\ &\leq P_{col}^{max} \end{aligned} \tag{2}$$

for any S', id' and cha'.

As a result, the Equation 1 and 2 contradict. So, the assumption cannot be true, i.e. there is no backdoor such that $P_{backdoor} > P_{col}^{max}$. And thus, the theorem is correct.

■