An Extended Case Study about Securing Smart Home Hubs through N-Version Programming

Igor Zavalyshyn, Nuno O. Duarte, Nuno Santos
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
{igor.zavalyshyn,nuno.duarte,nuno.m.santos}@tecnico.ulisboa.pt

Keywords: Smart Home, Internet of Things, Privacy and Security, Smart Hub, N-Version Programming.

Abstract: Given the proliferation of smart home devices and their intrinsic tendency to offload data storage and processing to cloud services, users' privacy has never been more at stake than today. An obvious approach to mitigate this issue would be to contain that data within users' control, leveraging already existing smart hub frameworks. However, moving the storage and computation indoors does not necessarily solve the problem completely, as the pieces of software handling that data should also be trusted. In this paper, we present a thorough study to assess whether N-version programming (NVP) is a valid approach in bootstrapping trust in these data handling modules. Because there are considerable complexity differences among the modules that process home environment data, our study addresses less complex modules that strictly follow exact specifications, as well as complex and looser modules which although not following an exact specification, compute the same high level function. Our results shed light on this complexity and show that NVP can be a viable option to securing these modules.

1 INTRODUCTION

In recent years, several smart home platforms have become mainstream, such as Samsung SmartThings, Apple HomeKit and Amazon Echo. However, the threat of privacy breaches constitutes a major source of concern for users. Device misconfiguration is frequent, which can lead to leakage of sensitive data, e.g., camera feeds (Kelion, 2012), or unauthorized home device monitoring (Forbes, 2013). Poor design and/or implementation of the software behind these devices is also a major security issue (Computerworld, 2016). SmartApps are often overprivileged and can abuse permissions to leak sensitive user data (Fernandes et al., 2016a).

A major difficulty in preventing unwanted sensor data exfiltration lies in the fact that many IoT applications, even if they were to execute entirely at the home environment, require both permissions to access sensor data (e.g., IP camera's frames) and to access the network. These permissions are required to allow the application to read and process the data, and send the results to the cloud. However, unless the application is correctly specified and implemented, its behavior can deviate from the expected, e.g., due to a bug, or an exploit, in order to release raw data over to the cloud, thus potentially causing a privacy breach.

Our goal is to investigate the adoption of N-version programming (NVP) as part of the design of smart hub platforms as a way to enhance security and prevent leaking raw sensor data to the cloud. Building on the shoulders of systems like FlowFence (Fernandes et al., 2016b) or Privacy Mediators (Davies et al., 2016), we consider a smart hub where IoT applications run and process sensor data locally under the constraint that applications cannot access such data directly, but through a mediation interface consisting of a set of *trusted functions* (TFs). TFs consist of extensions to the base hub platform that are implemented by third-party developers and that are deemed to correctly implement common data handling operations (e.g., face recognition, anonymization functions, etc.). The problem, however, is that if buggy or even malicious TF implementations are installed on the hub, serious security breaches can take place. NVP can help alleviate this problem by leveraging N different implementations of a single TF.

By using NVP, rather than depending on a single implementation, each trusted function depends on N different implementations (versions) that must concur to produce the final result. The smart hub feeds sensor data as input to each of the N function versions, and determines the overall output result based on a particular decision policy. For example, with total agreement policy, all partial outputs must be equal otherwise no output is released. A quorum policy requires only a quorum of equal partial responses to be reached. We envision different versions to be developed independently by an open community of developers. Insofar as the developers do not collude, N-version trusted functions are no longer dependent on the correctness of any specific function implementation as it is the case for existing smart hub solutions.

Although applying NVP to the smart hub architecture is relatively straightforward, the degradation of utility and performance can undermine the viability of this technique. The utility is penalized if an N-version module too often blocks any output to the application due to result divergence reasons. Performance of an N-version module tends to be bound by the slowest sub-module involved in the output decision. In our context, the impact to utility and performance will greatly depend upon how sub-modules are implemented. If sub-modules are developed from scratch, we expect most of the negative effects to be caused by implementation or performance bugs introduced by the developers. On the other hand, if sub-modules are built upon pre-existing code (e.g., libraries) such effects may also stem from incoherent specifications. The decision policy employed also plays a critical role in determining the behavior of modules.

In this paper, we provide an extended case study about the feasibility of NVP for securing smart home hubs. It seeks to characterize the impact of NVP to utility and performance of trusted functions. To this end, we perform an in-depth study focusing primarily on two main causes: software flaws and specification incoherence. We built multiple test modules performing a variety of privacy-sensitive functions, such as image blurring, voice scrambling, k-anonymization, face recognition, and speech recognition, among others. Then we tested them extensively in different N settings and under different decision policies.

Our in-depth study reveals that NVP has considerable potential for practical application within a smart home environment. In particular, we found that: (1) for N-versions that implement the same algorithm and follow the algorithm specification, it is possible to provide an N-module offering high utility as long as the number of software flaws is residual, (2) for N-versions that do not follow the same algorithm but perform the same task, we observe that although module utility can be negatively affected by output divergence, it can be increased leveraging decision policies tailored to the problem domain space, and (3) N-version trusted function module performance is typically bound by its slowest version, a condition that can be mitigated by leveraging versions redundancy.

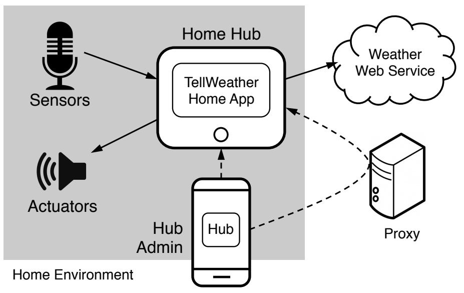

Figure 1: Appified privacy-preserving home hub.

Next, we provide a more extensive overview of our motivation, approach, and goals. In Section 3, we introduce a smart hub architecture based on NVP. Then, we present the main contributions of this work: a comprehensive study of the impact of NVP on TF utility (Sections 4 and 5) and performance (Section 6).

2 OVERVIEW

2.1 Privacy-Preserving Home Hubs

Figure 1 represents a privacy-preserving home hub platform (Davies et al., 2016; Fernandes et al., 2016b) in which security-sensitive sensor data can be aggregated and processed according to the privacy preferences of the user. The home hub is designed as an "appified" platform that allows for third-party developers to write *home apps* which users install on the home hub. In the figure, a home app called TellWeather waits for an audio command (e.g., "Tell weather in LA"), issues an HTTP request to a weather service, converts the response into audio signal, and forwards it to a speaker. The home hub provides an administration interface through which the homeowner can access the hub directly or tunneled through a proxy and manage it, e.g., install or uninstall apps, register devices, and set up privacy policies.

The hub platform provides app developers with API functions to interact with the devices. This API allows a home app to perform numerous operations, such as collecting data from sensor devices (e.g., audio from microphones, images from cameras), sending data to actuators (e.g., audio signal to speakers, or video streams to displays), accessing Internet services, and performing various data computations (e.g., speech or face recognition, or data anonymization). The operations that a home app is allowed to execute are controlled by a security policy: the home app must explicitly request the hub administrator for permissions to perform certain operations, in particular access to device APIs.

2.2 Trusted Functions: Goods and Ills

To prevent unlimited access to sensor devices, privacy-preserving home hubs allow their APIs to be extended with *trusted functions* (TFs) aimed to implement high-level operations that mediate access between the application and the raw data. In some cases a TF interposes between the application and a *data source*, e.g., a camera device. The motivation for such a TF can be, for instance, to provide a face recognition service over raw image data collected from the camera without revealing the raw data to client home apps. TFs can also mediate access to *data sinks*, for example to encrypt or anonymize sensitive data before sending it to a remote server. Some home hub solutions support TFs at data sources (Davies et al., 2016), others at data sinks (Mortier et al., 2016), and others in both (Fernandes et al., 2016b). Once installed into the hub, trusted functions can be invoked by local home apps running on the hub. TFs must be developed by third-parties and installed by the hub administrator. TF developers are fully trusted to implement them correctly. As long as TFs are correctly implemented, they constitute an effective approach to securely processing sensitive data. However, malicious TF implementations can perform serious attacks:

A1. Incomplete results: during processing, a malicious TF could intentionally omit parts of the results in an effort to disturb users' actions, e.g., hide the part "and B" when recognizing the user voice command "record game A and B".

A2. Incorrect results: similarly to the previous attack, a malicious TF could introduce incorrect results or replace correct with incorrect results, in order to trick the user into performing harmful operations, e.g., replace the name of the person the user wants to call with a premium number, when recognizing the user call request voice command.

A3. Data inferences: in collusion with a malicious application, a malicious TF could not only perform the operation it intended but also make inferences on the raw data and disclose it to the application, e.g., identify the people in the room in addition to recognizing the user voice command.

A4. Raw data leakage: the most devastating attack is the one where a malicious TF colludes with a malicious application and leaks raw data, e.g., send a raw camera frame as face recognition output.

2.3 Leveraging N-Version Programming

While the effects of attacks A1 and A2 can also stem from naive implementations, which are difficult to distinguish, we argue that attacks A3 and A4 are the sole product of lack of platform control over TF outputs. As a result we seek to understand whether relying on multiple TF implementations can mitigate these attacks. In particular, we aim to investigate the feasibility of N-version programming (NVP) to prevent malicious TF implementations from exfiltrating sensitive data outside of the home premises without the user's knowledge or consent.

TF implementations are expected to follow a TF specification. We assume that the TF specification is publicly available among home app developers and home hub users. As for a TF implementation, the TF binary needs to be publicly released, possibly even after being properly obfuscated. An NVP-based TF system must be able to detect the deviations in the functions outputs and react accordingly.

The N-version decision algorithm used to merge the outputs of multiple trusted function implementations must be efficient in terms of execution time and utility. Too strict algorithm will render the function useless, while the relaxed one might alter the security guarantees. Overall, the overhead introduced by employing N-version technique should not be significantly higher compared with a single version of trusted function execution.

Our main adversary consists of the potentially buggy or malicious code of a trusted function implementation. This implementation may try to output the sensitive user data as is without processing it but such a result will not be consistent across the outputs of all other implementations of the function, and will be ignored by the decision algorithm. We assume that various implementations of the same trusted function do not collude and are developed independently. We also assume that the software and hardware platform of the hub where the trusted function executes is secure, and that home apps and TFs execute in sandboxed environments. It is not our primary goal to secure against side-channel attacks. The capabilities of the attacker consist only of the ability to write arbitrary code as part of trusted function implementations.

3 TRUSTED FUNCTION MODULES

In this section, we present a general security architecture for smart home hub platforms based on N-version programming. In this architecture, home hub extensions consist of *N-version trusted function modules* (henceforth called "modules"). A module provides the functionality of a single TF implemented internally in a N-version fashion, with each of the N

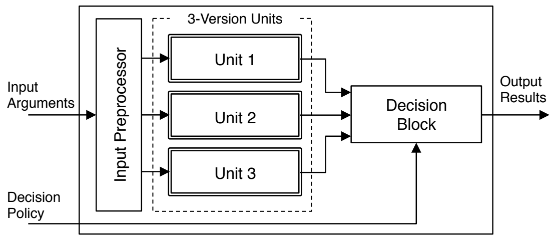

Figure 2: N-version trusted function module (with N=3).

versions being provided by independent developers. Each of these versions, called *units*, are required to implement the same *trusted function specification*.

Whenever an application issues a request, the input parameters are forwarded to all N units and their outputs are compared with each other before a final output is returned back to the application. Deciding whether or not a final output result is provided and what that output result will be depends on a *decision policy* defined by configuration. In a particular policy, all N units must produce the same result, which is then returned as output result, otherwise the application is informed that no result was generated. Thus, if any single unit implementation produces a malicious output, this output will differ from the remaining N-1 units (assuming no collusion) causing the final result to be suppressed, preventing the malicious unit from propagating its effects to the application.

Figure 2 shows the internals of a module implemented by 3 units. The input arguments are passed by the client application and the output results are returned to the application. The input preprocessor feeds the input arguments to each unit and the decision block implements a decision algorithm according to the provided decision policy. The decision policy is a configuration parameter decided by the hub administrator. Each unit is implemented by a program that runs in an independent sandbox. The input processor and the decision block logic must belong to the hub platform, which must also be responsible for setting up the units' sandboxes and the datapaths represented by arrows in Figure 2.

3.1 Module Lifecycle

The lifecycle of each module comprises four stages. In the *specification* stage, a cooperation between the platform and community developers results in the production and public release of module TF specifications. The decision on the creation of new modules is based on the community user needs. A specification features either the algorithm or high level function to be implemented, the input and output data formats, as well as a group of custom decision policies.

Once the specification is out, the module enters the *development* stage in which third-party developers independently implement their TF versions. This approach is similar to existing community-based software projects, e.g. Debian, where the members define task requirements and control the development process. Each TF version must be packaged and signed by the developer, and uploaded to the platform repository. By using a key that is certified by a certificate authority, it will be possible to assess the identity of the developer and prevent Sybil attacks, i.e., the same developer releasing and signing multiple malicious versions of the module's TF. Once authenticated the TF version is packaged in the TF module and subsequently either made available for users to install in case of a new module or automatically pushed for subsequent platform module update.

The next stage is *installation* of the module on the home hub. Users can download the latest version of the module from the repository and instantiate it locally on the hub. Default module settings work out of the box, however experienced users may add or remove module units, and redefine the decision policy according to their needs. Once the module is installed, the module enters the *execution* stage in which applications running on the hub are allowed to issue requests to the module. Note that modules may become temporarily out of service in order to perform software updates (e.g., installing a new unit or updating an existing one) and may also be permanently removed from the hub.

3.2 Detection of Unit Result Divergence

The decision taking process is at the core of what makes N-version programming effective at countering adversarial units. In the perfect scenario, each unit is assumed to execute one of two possible versions: *benign* or *adversarial*. A version is benign if it consists of a flawless implementation of the module's trusted function specification. A version is adversarial if it deviates from the intended specification in order to tamper with or leak sensitive data. Thus, if deviations exist between unit outputs, then at least one adversarial version is present. Since different security properties can be attained depending on the number of units in agreement, we define three decision policies providing three agreement conditions:

Total agreement (TA) policy: This policy offers the strongest security guarantees. All *N* units must agree on the same output result in order for an output to be returned. If this condition holds, the resulting value is returned, otherwise an error is yielded. Thus, 1 benign version only is required to exist in order to suppress the return of a corrupted result. In fact, for an attacker to be successful, all N versions must be both adversarial and collude in producing the same output.

Quorum agreement (QA) policy: Only a quorum $Q = \lfloor N/2 \rfloor + 1$ units (i.e., a majority) needs to reach consensus on a common return value. If Q is found, the module returns the agreed upon value, otherwise it reports failure. The QA policy is weaker than the TA policy because $Q > 1$ benign units need to be present to thwart an attack. Furthermore, a successful attack requires $Q < N$ colluding adversarial units.

Multiplex (Mux$_i$) policy: This policy is the weakest of all and can no longer be considered to provide N-versioning security benefits. Under a Mux$_i$ policy the decision block simply selects one unit output to be fed to the module output. The unit selection is parameterized by a number $1 < i < N$. This policy is useful mostly for debugging purposes during the testing stage of the module's lifecycle.

Ideally, the divergence between unit outputs in a module should occur due to the rational behavior of a malicious developer who intentionally had not implemented some version according to the trusted function specification of the module. However, other causes may lead to undesired output divergence that may cause undesired side-effects, namely: software flaws, and module incoherence.

3.3 Nondeterministic Inputs

One cause of unit divergence is *operational* and occurs whenever a specific trusted function depends on nondeterministic inputs, e.g., a random number, the system time or date, etc. If different units obtain different readings for the same intended input value, units' computations will likely return different results which may lead to failure in reaching a total or quorum agreement conditions and harm module's utility.

To avoid this problem, all nondeterministic inputs must be provided by the preprocessor. Sandboxes must prevent units from issuing nondeterministic system calls. If the version code depends on such calls, the input preprocessor can execute those upon request and pass the same value to all units. A request is declared by overriding the `init` method of the class of input parameters. The `init` method of this class is invoked by the input preprocessor and can be inherited by a subclass with the purpose of prefetching nondetermistic values. To prefetch an input value in a module, the trusted function specification only needs to assign this subclass to the type of the respective input argument. By constraining all units to receive the same input, this approach prevents the aforementioned operational causes for divergence.

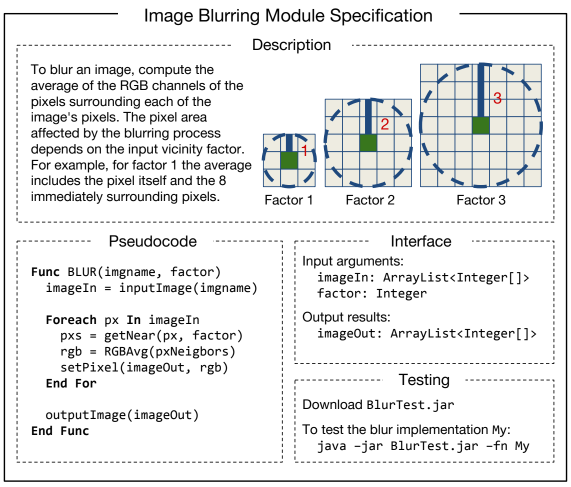
Image Blurring Module Specification

Description

To blur an image, compute the average of the RGB channels of the pixels surrounding each of the image's pixels. The pixel area affected by the blurring process depends on the input vicinity factor. For example, for factor 1 the average includes the pixel itself and the 8 immediately surrounding pixels.

Pseudocode

```
Func BLUR(imgname, factor)
  imageIn = inputImage(imgname)

  Foreach px In imageIn
    pxs = getNear(px, factor)
    rgb = RGBAvg(pxNeigbors)
    setPixel(imageOut, rgb)
  End For

  outputImage(imageOut)
End Func
```

Interface

Input arguments:
  `imageIn: ArrayList<Integer[]>`
  `factor: Integer`

Output results:
  `imageOut: ArrayList<Integer[]>`

Testing

Download `BlurTest.jar`

To test the blur implementation `My`:

```
java -jar BlurTest.jar -fn My
```

Figure 3: Image blurring module specification.

3.4 Software Flaws

A second unintended cause for internal result discrepancy is *accidental* in nature, and is caused by flaws in versions' software that cause the actual unit execution to deviate from the expected value as defined in the trusted function specification. In addition to harming module utility, flaws may negatively affect the correctness of the module. As shown in past studies, programmers tend to commit the same flaws in the same code regions, which may end up resulting in the generation of incorrect results that can eventually appear at the module's output depending on how many units have reached consensus on the same incorrect value and on the decision policy in place.

To reduce these negative effects, we define a format for trusted function specifications that aims to be both unambiguous and human readable so as to reduce the change of software flaws. Figure 3 depicts a simplified version of the specification for an image blurring trusted function. The specification format comprises: a *description* of the intended functionality, an *algorithm representation* in the form of pseudocode, the *interface* of the module indicating the input and output parameters and respective types, and a *testing procedure* which may include specific testing code. While the description and the algorithm representation aim to clarify misunderstandings about the specification, the testing parts aim to help debugging. Since the specification is public, the source code of the testing classes and types of input arguments / output results must be provided.

3.5 Module Incoherence

Module incoherence occurs if two or more units inside a module implement different trusted function algorithms. For example, a face recognition module may be based on software that implements face recognition using different techniques. As a result, one version may be able to identify a face that a second version cannot. Speech recognition is another example in which different algorithms may yield very diverse outputs, for instance being able to detect some words in a whole sentence, but not others.

A natural question that arises when the module is incoherent is whether it can be used for countering malicious version implementations. In fact, even assuming the absence of software flaws, it will be difficult to determine whether the divergence of results is due to a malicious version or due to semantic differences between versions themselves. Faced by this challenge, we take two decisions.

First, we require the modules must be explicitly specified as *strict* or *loose*. A strict module is one in which all versions must implement the same algorithm. For this reason, all versions are expected to strictly implement the algorithm described by the trusted function specification. In contrast, a module is loose if the implemented algorithm does not satisfy the specification. Version developers must clearly indicate the type of a given version. Otherwise, installing a loose version on a strict module will cause internal unit output divergence thereby severely degrading the module utility.

Second, to improve the utility of loose modules, we allow for replacing the standard decision algorithm of the decision block by a customized decision algorithm (which could be provided along with the trusted function specification). Since the standard decision algorithm simply tests the equality of units' outputs, algorithms that generate slightly different outputs will immediately fail the test which will considerably impair the module utility. On the other hand, a customized decision algorithm may perform domain-specific tests that may overcome small differences between outputs. The side-effect, however, is that by relaxing the equality requirement, an adversary may attempt to exploit that degree of freedom, e.g., to encode sensitive data to a remote party. Thus, by deciding whether or not to adopt a customized decision algorithm, an end-user can choose between the modules' utility and security.

Until now, we have presented an architecture for home hub based on N-version trusted function modules. We have also seen that the utility and security of each module can be affected by other factors, namely software flaws and module incoherence. The next sections focus on studying the impact of both these factors and on performance evaluation.

4 IMPACT OF SOFTWARE FLAWS

In this section we study the impact of version software flaws on the overall behavior of modules. We specifically focus on strict modules performance. Since they implement the same algorithm, it allows us to concentrate on discrepancies due to software faults. For our study, we implemented several test strict modules that feature common privacy-preserving algorithms for a smart home sensor data.

4.1 Experimental Methodology

We picked five different algorithms, and gathered three different implementations for each of them, with the help of five different volunteer developers. The versions for each algorithm were developed independently by different developers. For each developer, we provided a complete specification and a testing tool. The code was to be written in Java. Given the simplicity of the algorithms involved, we requested developers to submit their implementations before and after using the testing tool for debugging. While the implementations after testing recorded no bugs, the implementations before testing feature some bugs. Considering the purpose of this study, here we focus on the pre-testing implementations. The algorithms to implement were as follows:

Image blurring algorithm: An image blurrer can be used to protect users' privacy, namely by anonymizing the video data gathered by cameras (see Figure 3). We ran a simple battery test consisting of the blurring of 10 different pictures over vicinity factors of 1, 2 and 3. Afterwards, we made a byte-wise comparison between the expected result and the implementation produced files, in order to assess the implementations' correctness. In total, we executed 30 tests.

Voice scrambling algorithm: A voice scrambler can be useful in mitigating attempts to identify the speaker and other nearby individuals. This algorithm receives an audio clip as input, and after applying pitch shifting and distortion, it outputs a modified audio clip where the voice sounds robotized. With respect to testing, we exercised each implementation with 30 different audio clips.

Data encryption algorithm: RC4 is a stream cipher algorithm that can be used in encrypting certain home

Table 1: Evaluation results of strict modules under total agreement (TA) and quorum agreement (QA) decision policies. For each decision policy, the resulting output can be: correct (✓), incorrect (✗), or silent (–).

Module Function	Image Blurring			Voice Scrambling			Data Encryption			Data Hashing			K-Anonymization		
	V1	**V2**	**V3**	**V1**	**V2**	**V3**	**V1**	**V2**	**V3**	**V1**	**V2**	**V3**	**V1**	**V2**	**V3**
Single Tests Passed	$\frac{30}{30}$	$\frac{30}{30}$	$\frac{30}{30}$	$\frac{30}{30}$	$\frac{0}{30}$	$\frac{0}{30}$	$\frac{153K}{153K}$	$\frac{0}{153K}$	$\frac{153K}{153K}$	$\frac{41K}{41K}$	$\frac{41K}{41K}$	$\frac{0}{41K}$	$\frac{0}{210}$	$\frac{210}{210}$	$\frac{210}{210}$
Number of Bugs	0	0	0	0	4	4	0	1	0	0	0	1	1	0	0
N-mode Tests	TA: ✓, QA: ✓			TA: –, QA: ✗			TA: –, QA: ✓			TA: –, QA: ✓			TA: –, QA: ✓		

environment data before transferring it to a certain recipient. The final testing tool features 153K tests comprising tuples $\langle message, key, cyphertext \rangle$, where both *message* and *key* were randomly generated with increasingly longer sizes.

Data hashing algorithm: MD5 is a well-known hashing function useful in assessing the integrity of data. The final testing tool featured 41K tests. These tests consist of tuples $\langle message, hash \rangle$, where every *message* was randomly generated with increasingly longer sizes.

K-anonymity algorithm: Lastly, Mondrian is a top-down greedy algorithm for strict multidimensional partitioning, with the goal of achieving K-anonymity. Such an algorithm could be used in anonymizing home environment data (e.g., power consumption readings), so that the user could, for example, supply that information to an interested third party. The testing tool features 210 tests. These tests comprise tuples $\langle dataTuples, k, qids, result \rangle$, where *dataTuples* are statically grouped in 5 files each comprising 1 million entries, and *k* and *qids* are automatically generated and increased anonymity factors and quasi-identifiers respectively.

4.2 Main Findings

Table 1 summarizes the N-version study results, where V1, V2 and V3 correspond to three different version implementations. We highlight three main findings. First, under the TA decision policy, only the image blurring module yields an output. This is possible because all unit implementations passed the 30 tests. Since they produced the same result, the TA policy concurs on outputting the same result. This finding is consistent with the lack of bugs found in the code which could compromise the resulting output. For the remaining modules, however, faults have caused some versions to fail individual tests thus undermining the overall result.

Second, under the more relaxed QA decision policy, we observe that four modules can successfully reach a consensus and produce an output: the image blurring module—whose individual implementations output consistent results—and three additional modules in which two out of three implementations generate the same result, thereby allowing a consensus to be reached. In these cases, functional divergence occurred due to the existence of bugs. In the data encryption module, we identified a bug in V2 that consisted of a wrong value swap between two variables. Regarding the data hashing module, we detected one bug in V3 which was later found to be a variable poorly initialized. In the K-anonymization module, V1 contained a coding error stemming from a wrong pseudocode interpretation of the scope of a variable. Specifically, a global variable used by several functions was supposed to be initialized in a certain function, but V1's developer declared the variable as local to that function, leading to issues in the other functions handling it. Lastly, in one case, the voice scrambling module produced an incorrect response under QA. This happened because two versions, namely V2 and V3 experienced the same 4 bugs each. More specifically, the bugs originated from the wrong interpretation of a loop upper bound.

Given these numbers, we conclude that when versions yield different results, NVP actually detects (except for side-channels) implementation deviations created with rational intent. The exception being when the majority of the versions output the same erroneous result. Accidental mistakes can cause a reduction in the utility of the module. If a very conservative decision policy is employed (TA) this loss will be considerable (up to 80%). On the other hand, under QA, the utility drop is smaller, as four out of five modules can still produce the same result.

5 IMPACT OF MODULE INCOHERENCE

This section studies the impact of module incoherence on the modules' overall behavior and utility. For our study, we implement two test loose modules which do not strictly follow the same specification, yet compute the same high level function: *face recognition* and *speech recognition*.

Table 2: Success rates of face recognition (Recogn) measured in correct (✓), incorrect(✗) and no recognition (No Recogn).

		OpenCV	OpenBR	OpenFace	Decision Policy			MS Face API
					Total Agree.	OpenFace ∩ OpenBR	Quorum Agree.	
Recogn	✓	156 (≈62%)	219 (≈88%)	228 (≈91%)	137 (≈55%)	202 (≈81%)	220 (88%)	249 (≈99%)
	✗	1 (≈1%)	1 (≈1%)	0 (0%)	0 (0%)	0 (0%)	1 (1%)	0 (0%)
No Recogn		93 (≈37%)	30 (≈11%)	22 (≈9%)	113 (≈45%)	48 (≈19%)	29 (11%)	1 (≈1%)
Total		250 (100%)	250 (100%)	250 (100%)	250 (100%)	250 (100%)	250 (100%)	250 (100%)

The module implementing the face recognition function uses three existing open source face recognition libraries as building blocks: OpenCV (with Face module), OpenBR, and OpenFace. The libraries code remained unchanged but was wrapped around the N-version module's API. Based on these libraries, we defined several module configurations. We tested the effectiveness of the face recognition module when trained with a training set of 2250 images and a testing set of 250 images. In total, we trained the recognition of 250 different people with 9 pictures each. All these images where extracted from the UFI dataset. Microsoft Face API was used as state of the art face recognition implementation. It was trained and tested using the same dataset.

The speech recognition module uses three independent speech recognition libraries—Sphinx, Julius, and Kaldi—and was also tested in different module settings. Every configuration was exercised with 130 sentence tests from CMU's AN4 speech recognition dataset. As with face recognition libraries, we developed an API wrapper for all the speech recognition libraries. We use Google Speech API as state of the art speech recognition system which requires no training.

5.1 Face Recognition Module Study

Table 2 presents the success rate of our tests for the three face recognition functions evaluated individually, and the representative three module configurations, namely total agreement, quorum agreement and an intersection of the two functions that showed the best recognition results.

The first important observation is that the efficacy of the open source libraries is smaller than Microsoft Face's, which reaches 99% success rate. OpenCV stands out as the least effective library (only 62% success rate). The difference between OpenCV and OpenBR stems from the algorithms they implement, namely Eigenfaces and 4SF respectively. The small difference between OpenBR and OpenFace comes as a surprise, given that OpenFace implementation uses neural networks for face recognition, theoretically more effective than OpenBR's 4SF.

Table 2 then shows the success rate for three face recognition module configurations. Configuration *total agreement* consists of a module that employs all three libraries—OpenCV, OpenBR, and OpenFace—and yields "success" if and only if all libraries identify the same individual. Here we can see that the face recognition accuracy drops considerably to only 55%, which is explained by the significant differences that exist between the algorithms implemented by each library. In a second configuration, we used only two libraries—OpenFace and OpenBR—and in this case the success rate increased substantially to 81%. The best results were achieved when we used three libraries, but with a merging policy function that outputs success every time at least two libraries produce the same response. In this configuration (*quorum*), the success rate reaches 88%, which represents a reduction of only 3% when compared to OpenFace alone.

Considering these results, we argue that the best mechanism in merging face recognition results in an N-version setting is to gather the majority of the results given by a module's units. Note, however, that result intersection is not always a sound solution. If we consider the case where a module has fewer honest units than intentionally ineffective ones, e.g., units that produce wrong results with the goal of preventing face recognition, then the success and consequent effectiveness of the module is compromised. In order to address this issue, we believe a reputation based approach for unit selection could be used.

5.2 Speech Recognition Module Study

Although, word error rate (WER) is the metric generally used to measure the accuracy of speech recognition, it cannot be applied to the situation where there are multiple recognition results. Moreover, in a smart home scenario, voice commands can still be interpreted correctly even if some words are not recognized or come in a wrong order. We, therefore, opted for a sentence match and word intersection merging functions as the main performance parameters for speech recognition modules.

Table 4 shows the results for each library evaluated based on two criteria: *sentence match* and *word*

Table 3: N-version speech recognition confidence.

Decision Policy	Total Agreement	Sphinx ∩ Julius	Sphinx ∩ Kaldi	Julius ∩ Kaldi	Quorum Agreement
Sentence Match	$\frac{13}{130}$ (≈10%)	$\frac{13}{130}$ (≈10%)	$\frac{19}{130}$ (≈15%)	$\frac{34}{130}$ (≈26%)	$\frac{40}{130}$ (≈31%)
Word Intersection	$\frac{455}{902}$ (≈50%)	$\frac{455}{902}$ (≈50%)	$\frac{554}{902}$ (≈61%)	$\frac{557}{902}$ (≈62%)	$\frac{666}{902}$ (≈74%)
Word Union	$\frac{753}{902}$ (≈83%)	$\frac{706}{902}$ (≈78%)	$\frac{745}{902}$ (≈83%)	$\frac{735}{902}$ (≈81%)	$\frac{753}{902}$ (≈83%)

Table 4: Speech recognition confidence.

Implementation	Sphinx	Julius	Kaldi	Google
Sentence Match	$\frac{20}{130}$ (≈15%)	$\frac{36}{130}$ (≈28%)	$\frac{88}{130}$ (≈68%)	$\frac{103}{130}$ (≈79%)
Word Intersection	$\frac{578}{902}$ (≈64%)	$\frac{570}{902}$ (≈63%)	$\frac{719}{902}$ (≈80%)	$\frac{722}{902}$ (≈80%)

intersection. Sentence matching consists of the exact match between the entire original sentence and the recognized result returned by each library. Word intersection counts the number of words that exist in the original sentence and are also present in the recognition results returned by the library (902 is the total number of words present in all sentences). Table 4 shows that across both these dimensions, Sphinx and Julius clearly fall behind Kaldi, which offers the highest success rates (68% sentence match and 80% word intersection). At the same time, Kaldi' numbers are not far off Google Speech's.

Table 3 lists multiple module configurations that we used to produce speech recognition functions based on these libraries. Each entry of the table corresponds to a specific module configuration. The columns indicate which libraries constitute the units of the module, and the lines indicate the merging function that was used to produce a successful speech recognition output. We adopted three merging approaches: *sentence match*, which is similar to the criteria used for the individual solutions and issues an output if all units identified the same sentence; *word intersection*, which returns only the words that all units identified successfully; and *union*, which returns the union of all words identified by all units.

As shown in Table 3, *sentence match* tends to yield very poor results, displaying a success rate between 10% and 26% between any pair of units. Even when we consider quorum agreement, i.e., when at least two out of the three units return the same result, the success rate only reaches 31%, which is very far from Kaldi's 68%. Still, given that most speech controlled devices, e.g., Amazon Echo, use a grammar based approach, where they ask users to repeat words when they cannot recognize some, sentence match is an unreasonable speech recognition metric.

With word intersection, the results improve significantly up to 62% between any pair of units, and up to 74% when we consider the quorum for the results produced among them. Because of the intersective nature of the merging functions *sentence match* and *word intersection*, the adoption of an increasing number of units does not necessarily yield better results. This happens because the overall success rate is always bound to the performance of the worst units. This can be seen in the last column of the table. For instance, although the pair Julius and Kaldi yields a 62% success rate for the *word intersection* function, the addition of Sphinx bounds the three units overall success to the result yielded by the worst Sphinx pairing result, i.e., the result of the pair Sphinx and Julius (50%). The table also shows that for this type of functions the best approach is to use a quorum policy, i.e., the consensus between at least two units, which yielded success rates of 31% and 74% for *sentence match* and *word intersection* respectively.

Overall the highest success rate is achieved when word union is employed. As can be seen in the table, the function *word union* yields success rates of at least 78%, and 83% in the best case, surpassing even Google Speech. Contrary to *sentence match* and *word intersection*, the success rate of this function is the same for the combination of all three units and the quorum consensus (83%). This happens because quorum also implies the output of all three units. As a result, both functions produce the same output. Still, we argue that union is not a fair result merging function for two reasons. On one hand, semantically, the union of the output of two or more speech recognition units may differ significantly from a speech recognizer expected result. On the other hand, this union function can potentially endanger the privacy of the user. For instance, as long as there is one rogue unit that extracts information from the audio source, e.g., a voice detector that derives the number of people in the room based on the background sound, the whole module could be compromised, as its result would feature that information.

After analysing these numbers we can draw three conclusions: (1) exact sentence match is a poor speech recognition N-version result merging function, (2) word intersection recognition success rates are limited by the worst unit, but are reasonable when used in a quorum consensus approach, and (3) although word union success rates are the highest

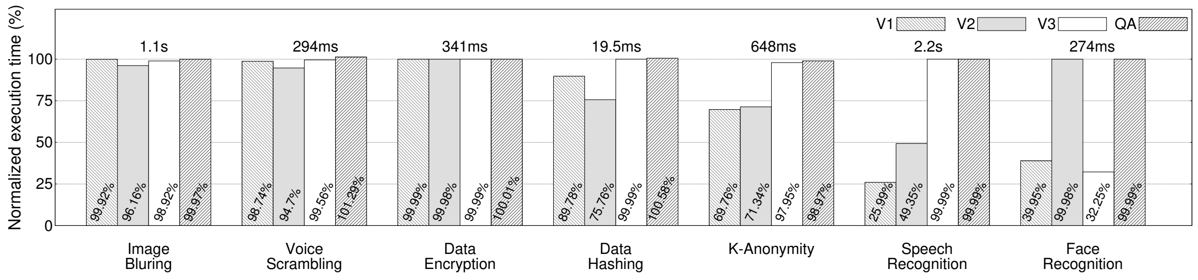

Figure 4: Strict and loose modules performance

among the configurations studied, its semantics and privacy limitations render it unusable in merging N-version results. Consequently, we argue that quorum-based word intersection is the best approach of the three in merging this type of results. Similarly to the face recognition case, it can also be complemented with a reputation based approach, in order to address the issue of the intentionally ineffective sub-modules.

6 PERFORMANCE EVALUATION

This section aims at assessing the performance overhead introduced by our approach as opposed to running a single instance of these algorithms.

6.1 Experimental Methodology

The performance evaluation comprises the execution time measurements of each of the aforementioned N-modules. These measurements feature the execution time of each of the three units comprising these modules, and the execution time of the quorum and total agreement merges. Each of these measurements consisted of computing the average of 50 tests, each with the same input. More specifically, we chose a 1280x720 pixel image and a factor of 2 for the image blurrer; a 10 second voice clip for the voice scrambler; a randomly generated 256-byte key and 1MB plaintext for the data encryption module; 1MB worth of randomly generated text for the data hashing module; and a set of 100000 tuples and a K-anonymity of 500 for the K-Anonymization module. For the face recognition module, we provided a training dataset of 150 pictures of three different people, and an additional picture as test input; and for speech recognition, we provided a general acoustic and custom language models as knowledge base, and a voice clip as input. The experiments were conducted on a laptop equipped with an Intel i3-3217U 1.80GHz CPU and 4GB of RAM. Similar computing resources can be provided by popular smart home hubs, e.g. Google OnHub or Google Home, that feature dual- or quad-core 1.5GHz CPUs with 512MB of RAM, which is enough for running multiple versions of TFs.

6.2 Main Findings

Figure 4 presents the performance results of the strict and loose modules. This figure shows the normalized execution time of each of the modules' units, as well as the two merging approaches. For a matter of consistency we take the TA policy as baseline. Note that the most significant performance differences among the different strict modules' units relate to either ineffective loop implementations, or recurrent use of data type casts. However, for the loose modules, the main performance difference stems from units' underlying algorithms diversity and implementations.

The first finding is the confirmation that the parallel execution nature of our approach bounds the two merging approaches' execution times to the slowest unit's execution time. This is most evident for the strict K-anonymization V3 unit. For loose modules the difference between unit execution times is even more noticeable. For the speech recognition module, V1's execution took a quarter of the time needed to execute V3. The same is observed for the face recognition module, where V3 outperformed V2.

Secondly, there is a significant execution time difference between loose module units. Note again that loose modules rely on heterogeneous versions. As a result, the underlying algorithms of units and their complexity may vary, leading to performance differences. Unlike strict modules, where the performance of units is usually similar, the impact of the slowest units on loose modules' performance is higher.

The third finding relates to the cost of the merging approaches. While we defined the TA policy as baseline to compare the performance of the three units and merging approaches, we can see that quorum agreement is sometimes more expensive than total agreement. This happens because, total agreement implies at most two comparisons, i.e., between V1 and V2, and between V2 and V3, while quorum agreement, in the worst case, requires three comparisons to yield a result. On the other hand, in the best case, quorum agreement can be achieved with one comparison only.

7 DISCUSSION

Traditionally, NVP has raised two main objections. First, N-version is regarded as demanding significant human resources to implement the N different software versions. However, considering our targeted scenario, this concern may be alleviated by relying on open source communities for the development of TF implementations. In fact, such communities have shown good results in maintaining large scale projects, e.g., Debian packages, python modules, and IoT specific ones, e.g., apps and automation recipes.

A second objection to NVP is the connotation of poor failure diversity among independent versions (Knight and Leveson, 1986). With this respect, it has also been shown (Knight and Leveson, 1986) that statistically, the number of common errors is relatively low and the diversity of implementations makes the overall system robust to failures. Therefore, it is hard for an adversary to exploit a common flaw across all the N-version modules. Although at a small scale, our software flaw study seems to confirm this idea, since in five different TFs, common flaws occurred only once. Even so, although this occurrence was detected by simple debugging tools, another reason behind it could be our specification effectiveness, which was not experimentally tested. Nevertheless, NVP considerably raises the bar for adversaries since the number of latent vulnerabilities would be smaller compared to single version executions.

Our approach's open source nature may also hinder TF utility, as the number of naive or malicious TF units outputting incorrect results may be higher than that of correct units. We propose two approaches to address this issue. First, a TF developer reputation scheme could provide insights regarding the effectiveness of a TF unit. This information could then be used to filter unwanted units when packaging modules. Second, at least for loose modules, their effectiveness could benefit from commercial software, which from our experience, requires little adaptation effort with our approach.

Performance-wise, the QA policy's positive results seem to suggest that the impact of the slowest unit for both loose and strict modules can be eliminated by taking advantage of unit redundancy. Instead of waiting for the slowest unit to finish, the decision block may process unit outputs up until a majority is formed. This approach addresses the performance problem and provides a reasonable tradeoff between module performance and user privacy.

As for malicious behaviour it is not in our scope to prevent malicious application attacks. This holds true for both attacks targeting hub security mechanisms, e.g., sandboxing, and TF module security, e.g., bug exploitation by sending crafted inputs to modules. Nevertheless, to address TF module security, our design could be complemented with unit address space randomization techniques (Cox et al., 2006).

8 RELATED WORK

NVP (Chen and Avizienis, 1978) has originally been used to reduce the likelihood of errors and bugs introduced during the software development. Multiple independent teams of programmers developed several versions of the same software and then ran these implementations in parallel.

Since then, NVP has been used in several fields. Veeraraghavan et al. (Veeraraghavan et al., 2011) propose multiple replicas of a program to be executed with complementary thread schedules to identify and eliminate data race bugs that can cause errors at runtime. DieHard (Berger and Zorn, 2006) uses randomized heap memory placement for each replica to protect the software from memory errors, e.g. buffer overflow or dangling pointers. Imamura et al. (Imamura et al., 2002) applies N-version programming in the context of genetics to reduce the number and variance of errors produced in genetic programming. Some systems (Cadar and Hosek, 2012; Giuffrida et al., 2013), apply N-version to the process of updating software, in order to detect and recover from errors and bugs introduced by the new versions. While these approaches assume there is only one developer of multiple software versions, we assume multiple independent developers and versions.

CloudAV (Oberheide et al., 2008) provides antivirus capabilities as a network service and leverages NVP to achieve better detection of malicious software. However, nothing prevents it from exploiting private user data. Demotek (Goirizelaia et al., 2008) employs N-version to enhance the reliability and security of several components comprising an e-voting system. Still, it assumes the modules are honest, and its main goal is to make it difficult for an attacker to compromise the whole system. Overall, none of the aforementioned systems rely on N-version to bootstrap trust in system components, focusing instead on improving reliability and availability.

Additionally, NVP has been used to detect and prevent system security attacks such as inadvertent memory access (Cox et al., 2006; Salamat et al., 2009). This, however, requires a custom memory allocation manager and modifications to the OS kernel. Moreover, these systems trust multiple versions of the same software and assume only the input data to be potentially malicious. NVP has also been leveraged to ensure personal information confidentiality and prevent information leaks. Most of these systems employ techniques in which two replicas of the same software are executed with different inputs (Yumerefendi et al., 2007), under different restrictions (Capizzi et al., 2008) or on different security levels (Devriese and Piessens, 2010). To the best of our knowledge, our work is the first to study the feasibility of NVP in securing smart hub platforms.

9 CONCLUSIONS

In this paper, we performed an extensive study on the use of NVP in order to enhance the security of TF-based smart hub platforms, which deal with home sensitive data. Our work comprises a thorough study on both strict and loose trusted function specifications. The results provide insights on our approach's effectiveness, and foster discussion surrounding utility, performance, and security issues associated with naive and malicious implementation output results.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their comments and suggestions. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via projects UID/CEC/50021/2013 and SFRH/BSAB/135236/2017.

REFERENCES

Berger, E. D. and Zorn, B. G. (2006). Diehard: probabilistic memory safety for unsafe languages. In *Proc. of PLDI*.

Cadar, C. and Hosek, P. (2012). Multi-version software updates. In *Proc. of ICSE*.

Capizzi, R., Longo, A., Venkatakrishnan, V., and Sistla, A. P. (2008). Preventing information leaks through shadow executions. In *Proc. of ACSAC*.

Chen, L. and Avizienis, A. (1978). N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. of FTCS-8*.

Computerworld (2016). Chinese Firm Admits Its Hacked Products Were Behind Friday's DDOS Attack. `http://www.computerworld.com/article/3134097`. Accessed May 2018.

Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. (2006). N-variant systems: A secretless framework for security through diversity. In *Proc. of Usenix Security*.

Davies, N., Taft, N., Satyanarayanan, M., Clinch, S., and Amos, B. (2016). Privacy Mediators: Helping IoT Cross the Chasm. In *Proc. of HotMobile*.

Devriese, D. and Piessens, F. (2010). Noninterference through secure multi-execution. In *Proc. of SP*.

Fernandes, E., Jung, J., and Prakash, A. (2016a). Security Analysis of Emerging Smart Home Applications. In *Proc. of SP*.

Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., and Prakash, A. (2016b). FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proc. of USENIX Security*.

Forbes (2013). When 'Smart Homes' Get Hacked. `http://www.forbes.com/sites/kashmirhill/2013/07/26/smart-homes-hack`. Accessed May 2018.

Giuffrida, C., Iorgulescu, C., Kuijsten, A., and Tanenbaum, A. S. (2013). Back to the future: Fault-tolerant live update with time-traveling state transfer. In *Proc. of LISA*.

Goirizelaia, I., Selker, T., Huarte, M., and Unzilla, J. (2008). An Optical Scan E-Voting System Based on N-Version Programming. *IEEE Security & Privacy*, 6(3):47–53.

Imamura, K., Heckendorn, R. B., Soule, T., and Foster, J. A. (2002). N-Version Genetic Programming via Fault Masking. In *Proc. of EUROGP*.

Kelion, L. (2012). Trendnet security flaw exposes video feeds. `http://www.bbc.com/news/technology-16919664`. Accessed May 2018.

Knight, J. C. and Leveson, N. G. (1986). An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, pages 96–109.

Mortier, R., Zhao, J., Crowcroft, J., Wang, L., Li, Q., Haddadi, H., Amar, Y., Crabtree, A., Colley, J. A., Lodge, T., Brown, T., McAuley, D., and Greenhalgh, C. (2016). Personal Data Management with the Databox: What's Inside the Box? In *Proc. WCAN CoNEXT*.

Oberheide, J., Cooke, E., and Jahanian, F. (2008). CloudAV: N-Version Antivirus in the Network Cloud. In *Proc. of USENIX Security*.

Salamat, B., Jackson, T., Gal, A., and Franz, M. (2009). Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. of EuroSys*.

Veeraraghavan, K., Chen, P. M., Flinn, J., and Narayanasamy, S. (2011). Detecting and surviving data races using complementary schedules. In *Proc. of SOSP*.

Yumerefendi, A. R., Mickle, B., and Cox, L. P. (2007). Tightlip: Keeping applications from spilling the beans. In *Proc. of NSDI*.