Tamper Resistance for Software Defined Radio Software

Shucai Xiao, Jung-Min “Jerry” Park, and Yanzhu Ye
Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA, 24061, USA
Email: {shucai, jungmin, yeyz}@vt.edu

Abstract

The security of software defined radio (SDR) software is essential to the trustworthiness of the overall radio system. When designing and developing security solutions for SDR software, its performance requirements, such as stringent real-time constraint, need to be considered. In this paper, we describe a tamper resistance scheme that was designed to thwart the unauthorized tampering of SDR software. This scheme utilizes code encryption and branch functions to obfuscate the target program\(^1\) while enabling the program to satisfy its performance requirements. The scheme employs a technique called the Random Branch Function Call (RBFC), which enables a user to control the tradeoff between integrity checking frequency and the overhead. We have rigorously evaluated the scheme using various performance metrics and quantified the relationship between the end-to-end delay overhead (caused by the tamper resistance scheme) and voice quality in the context of a voice communication network.

1 Introduction

The flexibility and adaptability brought by modern software, low-cost microprocessors, and smart antennas have made Software Defined Radios (SDRs) a reality. Unlike conventional radios, an SDR can readily alter its operating parameters, such as frequency, modulation method, and output power (either radiated or conducted) [20]. Such operating flexibility is possible because the core functionalities of an SDR are carried out by software instead of hardware/firmware. However, the advantages of SDRs can be offset by the lack of security and reliability of the underlying software. We expect that the emergence of SDRs will bring about new security threats that have not been considered previously. The programmability of SDR devices may enable adversaries to carry out security violations such as unauthorized radio software downloads. An even more serious threat is the possibility that adversaries may attempt to manipulate radio software to gain operational advantages (e.g., transmit at a power higher than the authorized limit or on the wrong frequency) or launch attacks against incumbent networks. The prospect of unauthorized changes to SDRs’ operating characteristics (i.e., power, frequency, and modulation) is a major concern for regulators and wireless service providers. Hardening the radio software against unauthorized tampering is one way of thwarting malicious users from exploiting the programmability of SDR to launch attacks.

In this paper, we propose a novel tamper resistance scheme for protecting SDR software. Our work is complementary to current software tamper resistance techniques, but, at the same time, is unique in the sense that it is specifically designed for radio software with stringent real-time constraint. To the best of our knowledge, this work represents the first attempt at addressing the important problem of SDR software tampering. The proposed scheme thwarts illegal modification of the target software by making binaries very difficult to disassemble. The scheme makes an appropriate tradeoff between the degree of tamper resistance and the performance overhead to enable robust protection while meeting the software’s performance requirements.

The scheme’s noteworthy characteristics include the following: 1) To obscure control flows, it transforms unconditional branch instructions to calls to a branch function; 2) To provide robust protection while minimizing overhead, it encrypts only the target addresses of the transformed branch instructions (note that previous encryption-based approaches [21] encrypted entire blocks of code and, as a result, incurred high performance overhead); 3) It uses constants called address obfuscation values to provide enhanced resilience to static disassembly; and 4) It employs a technique called the Random Branch Function Call (RBFC), which enables the user to control the tradeoff be-

\(^{1}\)In the rest of this paper, “software” and “program” will be used interchangeably.
between the degree of tamper resistance and the overhead by modulating the frequency of integrity checks.

The rest of the paper is organized as follows. The related work is described in Section 2. The detailed description of the proposed scheme and its security analysis are given in Sections 3 and 4, respectively. In Section 5, we evaluate the scheme’s performance and present experimental results. Section 6 concludes this paper.

## 2 Related work

The work presented in this paper is most closely related to techniques for software obfuscation, software tamper resistance, and attacks to software protection schemes. In the following paragraphs, we briefly describe current techniques for obfuscation, tamper resistance, and attacks to the software protection schemes.

Obfuscation transforms software into a new form, which has the same functionality as the original one, but its data structure and control flow graph are more complex and difficult to understand than the original software. Obfuscation can be implemented in two ways: obfuscating code instructions or obfuscating program control flow. Code instruction obfuscation is implemented by inserting “junk bytes” into the program [13] or encrypting the instructions of the program [21]. In control flow obfuscation, the control flow graph of the program is obfuscated using techniques such as branch functions [13] and opaque predicates [4], which make the protected software more resistant to reverse engineering. Working from the opposite viewpoint, a number of researchers have devised attacks against obfuscated systems. For instance, in [11], Jacob et al. describe an attack against an obfuscated Data Encryption Standard (DES) cipher by injecting faults to obtain the decryption key.

Tamper resistance involves techniques to detect or thwart integrity violations of software. Integrity checking is one of the most commonly used methods in implementing tamper resistance. Most schemes of this type calculate the hash values of portions of the software and use those values in one of several ways to detect or deter integrity violations. One simple method for tamper resistance is comparing the calculated hash values to their corresponding pre-computed values [8]. Another method involves using hash values of code blocks as keys to encrypt the entire program [21] or parts of it [2]. In another approach, hash values are used to compute the correct target addresses of branch instructions within a branch function [12]. There are also other types of tamper resistance schemes. For example, in [1], the authors proposed a scheme that uses a monitor process to monitor the execution of the software. In [16], a scheme using a finite state machine (FSM) is proposed in which the actual instruction to be executed is dependent on the FSM state and the instruction fetched from the protected software. It is worth noting that a number of techniques exist to circumvent certain types of tamper resistance. For instance, in [22], Wurster et al. propose an attack that exploits the difference between data reads and instruction fetches to circumvent checksumming-based tamper resistance schemes.

## 3 Scheme Description

### 3.1 Overview

The proposed tamper resistance scheme uses a combination of code encryption and a branch function to obfuscate the control flow of a program while enabling the program to meet its performance requirements. The scheme replaces selected unconditional branch instructions with calls to a branch function. To minimize overhead, only the target addresses of the selected branch instructions are maintained in encrypted form, instead of encrypting entire blocks of code. Hash values of code blocks are used as the encryption keys. Integrity violations of the program would cause hash values to be different from the original ones, thus resulting in incorrect decryptions of the target addresses which in turn leads to erroneous software execution.

The general idea of a branch function is illustrated in Fig. 1. The code of a branch function can determine the target addresses, $A_i (i = 1, 2, \cdots, N)$, of branch instructions and branch to the appropriate target addresses. Its formal definition is given in [13]. A branch function serves two important purposes in our scheme. First, the branch function obfuscates the software control flow by obscuring how the target addresses are computed. Second, code integrity checking is performed within the branch function.

The proposed scheme is implemented at the assembly code level. To apply a tamper resistance scheme, we need to first compile the source code into assembly code, then implement the tamper resistance scheme in the assembly code, and finally assemble the software into binary code.

There are two processes that dictate how our tamper resistance scheme works to protect a given program: transformation and verification processes. The transformation process is the process of implementing the tamper resistance scheme offline in the “unprotected” assembly code. The verification process refers to the process where the tamper resistance scheme works to protect a given program: transformation and verification processes. The transformation process is the process of implementing the tamper resistance scheme offline in the “unprotected” assembly code. The verification process refers to the process where the tamper resistance scheme works to protect a given program: transformation and verification processes.
per resistance scheme checks for integrity violations as the “protected” binary code executes.

There are several existing obfuscation methods and tamper resistance schemes that also utilize an encryption-based mechanism [21]. Our scheme, however, integrates encryption-based tamper detection with other tamper resistance techniques in a distinctive fashion that makes an appropriate tradeoff between tamper resistance and overhead. These techniques include: 1) encrypting only the target addresses of selected branch instructions, 2) use of Address Obfuscation Values (AOVs) to provide enhanced resistance to static disassembly, and 3) the RBFC technique which enables users to tune the integrity checking frequency. Each technique is described in detail in the following subsections.

3.2 Software Transformation and Execution

3.2.1 Transformation Process

In the proposed tamper resistance scheme, the branch instructions are transformed to calls to a branch function and their target addresses are encrypted and stored in the data section of the program. Fig. 2 shows an example of this transformation process. Fig. 2 (a) illustrates the transformation of the assembly code instruction. The code instruction `jmp A1` in line 7 is transformed to the code instruction `call _branch`. The decrypted target address is calculated, the target address is retrieved from the table using the index value $i$, whose value will be used to read $C_i$ and the block starting and ending addresses $(A_{i,1}, A_{i,2})$ in the verification process. Here, $i$ can be any number between 1 and $N$. It is hard-coded and should be different for each branch instruction transformation.

For each selected branch instruction steps b) through d) are repeated.

3.2.2 Verification Process

The integrity checking is performed in the following steps:

a) When the software’s execution reaches the location of a branch function call, the hard-coded $i$ value, and the pair $(A_{i,1}, A_{i,2})$ are used as the input parameters for executing the branch function. The pair $(A_{i,1}, A_{i,2})$ is retrieved from a table using the index value $i$.

b) After software execution is transferred to the branch function, the hash value of the code block corresponding to the pair $(A_{i,1}, A_{i,2})$ is calculated. This value, $v_i$, is the key for decrypting the target address of the branch instruction associated with index $i$.

c) The encrypted target address $C_i$ is retrieved from the target address table $T$ by reading the contents of the table entry indexed by the value $i$.

d) The encrypted target address $C_i$ is decrypted using $v_i$. The decrypted target address is $A_i$.

e) The return address of the branch function is replaced with $A_i$. This will cause the program execution to resume at address $A_i$ after the completion of the branch function’s execution.
3.3 Enhanced Resistance to Static Disassembly

To make the proposed scheme more robust to static disassembly, a technique to obfuscate the starting and ending addresses of the code blocks is proposed. Recall that having knowledge of the starting and ending addresses of the code blocks is vital for successful reverse engineering of the program since those addresses are needed to correctly decrypt the target addresses of the branch instructions replaced by calls to the branch function. In the proposed tamper resistance scheme, the actual starting and ending addresses of the code blocks are not stored, but instead their obfuscated form are stored (in the data section of the code). The actual address values are calculated at runtime and then passed to the branch function when the branch function is called. The actual address is calculated by taking the exclusive OR (XOR) of the corresponding obfuscated address and the appropriate address obfuscation value (AOV). For instance, the starting address, $A_{1,1}$, (of the $i$-th code block) is computed at runtime as $A_{1,1} = x_{i,1} \oplus A_{i,1}$, where $A_{i,1}$ is the obfuscated address and $x_{i,1}$ is the AOV.

An AOV is initialized to 0 and then is incremented to the right value just before the corresponding call to the branch function is executed so as to ensure that the correct starting/ending address is passed to the branch function. The AOV is incremented to the correct value using several increment instructions, and those instructions are distributed throughout the assembly code in a random fashion. Without actually executing the program, it is impossible to obtain the correct starting and ending addresses of the code blocks (without having intimate knowledge of the program) because of the way they are calculated at runtime. Therefore, added resilience against static attacks is provided.

Algorithm 1 RBFC: An algorithm for tuning the frequency of branch function calls.

**Input:** $N_{\text{max}}$.

1: $\text{counter} \leftarrow 1$, $N \leftarrow 1$
2: $\text{TAddr}$;
3: if $\text{counter} < N$ then
4:     $\text{counter} \leftarrow \text{counter} + 1$
5:     $\text{Addr} \leftarrow \text{Ob}_{\text{Addr}} \oplus (\text{counter} + N)$
6:     $\text{Ob}_{\text{Addr}} \leftarrow \text{Addr} \oplus (\text{counter} + N + 1)$
7:     $\text{Jump Addr}$
8: end if
9: $\text{counter} \leftarrow 0$
10: $N \leftarrow \text{random}(N_{\text{max}})$
11: $\text{Ob}_{\text{Addr}} = \text{branch}(N, \cdots)$

3.4 Random Branch Function Call

The overhead of the proposed scheme is caused by the execution cost of the branch function calls. The amount of overhead is proportional to the number of branch function calls. To keep the execution cost acceptably low, we can adopt two techniques. One technique is to transform into branch function calls only those branch instructions that are executed less frequently. This technique was proposed by Linn and Debray [13]. The other technique is to decrease the number of branch function calls. When done properly, either technique can reduce the runtime overhead without causing notable degradation in robustness of tamper resistance. We propose a technique called Random Branch Function Call (RBFC) that enables a user to adjust the number of branch function calls. The RBFC technique is described in Algorithm 1 as pseudo code. In the algorithm, $N$ is a random variable with a discrete uniform distribution whose probability mass function is $P_r(N = i) = \frac{1}{N_{\text{max}}}$ for $i = 1, 2, \cdots, N_{\text{max}}$; random ($\cdot$) is a random number generator that generates numbers between 1 and $N_{\text{max}}$; $\text{branch}(N, \cdots)$ is the branch function; $\text{Addr}$ is the target address of a branch instruction and is stored in a register (e.g., eax); and $\text{Ob}_{\text{Addr}}$ is the obfuscated form of the target address. The algorithm specifies that the branch function is called only when $\text{counter} \geq N$; otherwise, $\text{counter}$ is increased by 1 and the program execution jumps to the target address, $\text{Addr}$, directly via the instruction $\text{Jump Addr}$ in line 7. To protect $\text{Addr}$ and make it impossible to be obtained via static disassembly, $\text{Addr}$ is calculated at runtime, i.e., just before the jump instruction executes, $\text{Addr}$ is calculated to be used. Then it will be set to an invalid value, which makes the actual value of $\text{Addr}$ exist for a very short time. The calculation of $\text{Addr}$ is via the XOR operation from $\text{counter}$, $N$, and $\text{Ob}_{\text{Addr}}$, of which, $\text{Ob}_{\text{Addr}}$ is calculated and obfuscated in the branch function via decryption. Furthermore, for each loop, $\text{Ob}_{\text{Addr}}$ is updated, which makes it even more difficult to be analyzed to obtain the correct value of $\text{Addr}$.

When RBFC is employed, the expected number of branch function calls in lieu of executions of a particular branch instruction is $(2N_0 - 2)/N_{\text{max}} + 1$, where $N_0$ is the number of times that the branch instruction would have executed without RBFC.

4 SECURITY ANALYSIS

In an attack against a tamper resistance scheme, an adversary attempts to circumvent any tamper detection and/or deterrence scheme (e.g., integrity checking, obfuscation mechanisms, etc.) that may be in place in order to reverse engineer and modify the target software. Generally, there are two types of attacks against a tamper resistance scheme:
static attacks and dynamic attacks. A static attack is solely based on the static information obtained from examining the software code and reasoning about possible behaviors without actually executing the software. For a dynamic attack, it is based on dynamic information observed while executing the software. The proposed scheme was designed to thwart static attacks and also provides some protection against dynamic attacks.

In a typical static attack, an adversary uses tools to disassemble the software, obtain code instructions and the control flow graph, and then circumvents the tamper resistance scheme after analyzing the assembly code. The proposed tamper resistance scheme thwarts this attack approach mainly in two ways. First, the proposed scheme makes the control flow construction of the entire program infeasible by encrypting target addresses of selected branch instructions (which are replaced by calls to the branch function) and then decrypting them during runtime. Second, the encryption keys are protected using AOVs whose values are very difficult to obtain from static attacks.

In a dynamic attack, instrumentation (e.g., the DIOTA [14]) is often used to record the software execution state needed for circumventing software protection mechanisms. In the context of the proposed scheme, software execution state may include information such as hash values of those protected code blocks. After getting execution state information, an attacker would likely attempt to use them to decrypt the target addresses and then convert the program back to its original form. However, there are several practical challenges in performing a successful dynamic attack. First, instrumentation code needs to be inserted into the program to record the execution state information which is difficult to carry out. Successfully carrying out this approach is unlikely because inserting instrumentation code will cause the hash values of the code blocks, which are calculated at runtime, to get corrupted. Corruption of the hash values will lead to the incorrect decryption of the target addresses, which in turn leads to program execution failure. Note that an adversary may try to circumvent the integrity checking mechanism altogether by exploiting the separation of code and data accesses prevalent in modern processors, such as the technique described in [22]. Fortunately, there is a relatively easy fix to thwart such an attack [6]. Second, even if the adversary were able to successfully insert instrumentation code, (s)he would need to identify the branch function calls before the tamper resistance scheme can be circumvented or removed. This requires the adversary to construct the inputs that precisely trigger the execution paths that the branch function calls are on. Since the branch function calls are spread out over different execution paths, it would be extremely difficult for the adversary to create the right set of inputs without having intimate knowledge of the protected software.

5 EXPERIMENTAL EVALUATION

5.1 Overview

We carried out multiple experiments to evaluate the performance of the proposed scheme. The experiments were performed in four parts: (1) Evaluation of the RBFC technique; (2) Measurements of overhead caused by the scheme (executable file size increase, runtime overhead, and end-to-end delay overhead); (3) Breakdown of the contribution of each type of operation to the runtime and end-to-end delay overhead. Since the execution cost incurred by operations such as branch function calls, address XOR operations, random number generation, etc, are insignificant compared to other operations, we lumped their contributions into a single item. So we classified all operations into three categories: hash value calculation, target address decryption, and other operations; (4) Tamper resistance scheme’s impact on voice quality in voice communications.

5.2 Experiment Settings

Our experiments were performed on the GNU Radio software [7] and the SPECINT2006 benchmark suite [19]. GNU Radio is a software toolkit for building software defined radio systems. Specifically, we chose to use a GNU Radio system that is composed of a simple Gaussian Minimum Shift Key (GMSK) transmitter and receiver. The actual signal transmission and reception were handled by universal software radio peripheral (USRP) boards. Our GNU Radio system implements the physical layer of a data transmission system and includes the following signal processing blocks: bytes to sample, Gaussian interpolate finite impulse response (FIR) filter, frequency modulator, frequency demodulator, clock recovery, and binary slicer (the first three blocks are for the transmitter and the rest are for the receiver). In our experiments, the tamper resistance scheme was applied only to the transmitter. In the SPECINT2006 benchmark suite, the components “Mcf”, “Sjeng”, and “Hmmer” were chosen to perform the experiments. We used the cipher algorithm XTEA [17] with the recommended round number 64 for encryption tasks and the hash function MD5 for the calculation of the code blocks’ hash values. Each numerical result presented in this paper is the result of repeating the same experiment six times and calculating the average of the six results.

5.3 Performance Metrics

5.3.1 Effectiveness of the RBFC technique

To test the effectiveness of the RBFC technique, the relative runtime and end-to-end delay overhead are compared
by setting different $N_{\text{max}}$ values. In our experiments, $N_{\text{max}}$ is set to $9, 19, 39, 99, 199,$ and $399$. Since $N$ is a random variable with uniform distribution, the expected number, $\bar{N}$, of branch instruction executions before a branch function call are $5, 10, 20, 50, 100,$ and $200,$ respectively.

### 5.3.2 Executable file size increase

To implement the proposed tamper resistance scheme, the following components need to be inserted into the program: branch function, hash function, decryption function, a table of starting and ending addresses of the code blocks, and a table of the encrypted target addresses. These components contribute to the software’s executable file size increase.

### 5.3.3 Runtime overhead

The relative runtime overhead, $O_t$, is defined as the additional execution time caused by the tamper resistance scheme relative to the execution time of the original program. It is calculated as $O_t = (T_{tr} - T_o) / T_o$, where $T_{tr}$ is the execution time needed to process a given input file with the tamper resistance scheme applied and $T_o$ is the execution time of the original software (without any tamper resistance) to process the same input.

### 5.3.4 End-to-end delay overhead

A SDR is a real-time system, and its software needs to satisfy stringent requirements in terms of end-to-end delay. We define the end-to-end delay overhead, $O_e$, as $O_e = (E_{tr} - E_o) / E_o$, where $E_{tr}$ is the end-to-end delay with the tamper resistance scheme applied and $E_o$ is the end-to-end delay of the original software.

### 5.4 Experiment Results

#### 5.4.1 Effectiveness of the RBFC technique

In this experiment, we recorded the relative runtime and end-to-end delay overhead of the GNU Radio system caused by the scheme as shown in Fig. 3. From Fig. 3, we can see that as $\bar{N}$ increases, the runtime and end-to-end delay overheads decrease quickly. From these results, we can see that RBFC is effective in decreasing the runtime and end-to-end delay overhead.

In the following experiments, we set $N_{\text{max}} = 399$ for all of the results obtained in the GMSK system.

#### 5.4.2 Executable file size increase

Table 1 shows the executable file size increase of the target software. From Table 1, we can see that the executable file size increase is small. For the GNU Radio program, it is less than 1% (“GMSK” refers to the GNU Radio program). For the programs in the SPECINT2006 benchmark, the size increase is less than 5% for all of the programs.

#### 5.4.3 Runtime overhead

In Table 2, the runtime overhead is shown. Note that the execution time of the GNU Radio program is dependent on the size of the input data file. From the data presented in Table 2, we can see that the runtime overhead caused by the proposed scheme is significantly less than that caused by existing tamper resistance schemes based on code encryption, such as the one proposed in [21].

In Fig. 4, we show the contribution of each type of operation to the overall runtime overhead. These operations are hash value calculations, target address decryption, and other operations. It is interesting to note that the integrity checking operation (i.e., hash value computations of code blocks) is the single greatest contributor to the runtime overhead, contributing over 50% for all of the programs tested.

#### 5.4.4 End-to-end delay overhead

The end-to-end delay overhead was measured only in the GNU Radio software. The average end-to-end delay of the original software was 33.04 ms. The average end-to-end delay increased to 34.35 ms when the tamper resistance was implemented—this represents an increase of 1.30 ms and the relative overhead is 3.93%. In those measurements, $N_{\text{max}}$ was set to 399. The contribution of each type of operation to the end-to-end delay is shown in Fig. 5. This result is similar to the one shown in Fig. 4.

#### 5.4.5 Impact on voice quality

The increase in execution time and end-to-end delay caused by the implementation of a tamper resistance scheme cannot be avoided. These performance criteria are critical to a real-time system, such as a SDR system. In this subsection,
we evaluate the proposed scheme’s impact on voice quality. In the past, different voice quality measurement techniques have been proposed to measure voice quality. These techniques can be classified into two types: subjective voice quality measurements, e.g., the Mean Opinion Score [15] and objective measurements, such as the R-factor [3]. In this paper, we use the R-factor.

According to [3], the R-factor can be calculated as

$$ R = 94.2 - I_e - I_d, $$ (1)

where $I_e$ is the impairment caused by packet loss and $I_d$ is that caused by delay. Experiments have shown that $I_e$ can be approximated by the following equation [3]:

$$ I_e = \lambda_1 + \lambda_2 \ln (1 + \lambda_3 e), $$ (2)

where $e$ is the total loss probability and the values of $\lambda_1, \lambda_2, \lambda_3$ depend on the type of the pack loss (random or bursty) and the voice coding method (G.729a [10] or G.711 [9]). For example, $I_e$ can be calculated as:

$$ I_e (G.729a, random) = 11 + 40 \ln (1 + 10e) $$ (3)

for the codec method G.729a and random packet loss.

The impact of delay can be similarly modeled as

$$ I_d = 0.024d + 0.11 (d - 177.3) H (d - 177.3) $$ (4)

where $H (\cdot)$ is a unit step function, $d$ is the end-to-end delay value. Besides the packet loss and the end-to-end delay, we also considered the delay variation (i.e., jitter), which can cause packet loss if the de-jitter buffer in the receiver side is not large enough. In [3], Cole and Rosenbluth expressed the relationship between the packet loss rate and the de-jitter buffer size as:

$$ e_{de-jitter\_buffer} = P \{l > b \cdot g\} $$ (5)

where $e_{de-jitter\_buffer}$ is the packet loss rate corresponding to the delay $b \cdot g$ caused by the de-jitter buffer, $g$ is the mean value of the inter packet arrival time, and $l$ is the inter packet arrival time of individual packets. As shown in [3], by using the Chebyshev’s inequality, the upper bound of the packet loss caused by the jitter buffer can be derived as:

$$ e_{de-jitter\_buffer} < \frac{v}{(b \cdot g - g)^2} $$ (6)

where $v$ is the variance of $l$.

To evaluate voice quality, the system architecture in Fig. 6 was considered. Since we want to isolate the impact of the tamper resistance scheme on voice quality, we assume a constant end-to-end delay and zero packet loss in the Network—i.e., $t_2$ is a constant value and no packet loss is caused by the Network. As a result, only the packet loss incurred by the de-jitter buffer in the receiver is calculated. The effect of the delay values, $t_1$ and $t_3$, and their variance are considered. According to [18], the value for $t_2$ is between 80ms and 170ms; we will set this value to 80 ms.

In evaluating the impact on voice quality, we assumed that the uplink delay and downlink delay are 72.1 ms and...
71.8 ms, respectively (these values are typical in a TDMA system [5]). The relative end-to-end overhead—3.93%, 8.58%, 27.16%, 33.51%, 56.91%, and 112.52%—tested in the testbed will be used to calculate the delay and jitter increase. After calculating the end-to-end delay and jitter, by specifying different de-jitter packet loss rates—$10^{-3}$, $5 \times 10^{-3}$, $10^{-2}$, $2 \times 10^{-2}$, $5 \times 10^{-2}$, and $10^{-1}$, the voice quality of the network shown in Fig. 6 can be calculated. In Fig. 7, the voice quality lower bound (i.e., R-factor) versus the end-to-end delay overhead is plotted for various packet loss rates. In Fig. 7(a), the G.729a codec and the random packet loss case is considered. Figs. 7(b) and (c) show the plots when the G.711 codec is considered, with the former for random packet loss and the latter for bursty packet loss.

From Fig. 7, the following observations can be made: 1) As expected, increase in end-to-end delay caused the R-factor lower bound to decrease. However, as can be seen from the figures, the packet loss rate had a more direct impact on the R-factor lower bound compared to end-to-end delay; 2) For a specific value of the end-to-end delay overhead, there is a tradeoff between the de-jitter buffer size and the de-jitter packet loss rate. A larger de-jitter buffer size reduces the packet loss rate, but causes an increase in end-to-end delay. An appropriate tradeoff between the packet loss rate and the end-to-end delay can be made by adjusting the de-jitter buffer size, which is needed to increase the voice quality. For example, in Fig. 7, when the relative overhead is 3.93%, better voice quality can be obtained by setting $e = 10^{-3}$ rather than setting $e = 10^{-2}$. But if the relative overhead is 8.58%, better voice quality can be achieved by setting $e = 10^{-2}$ rather than setting $e = 10^{-3}$.

6 Conclusions

In this paper, we proposed a tamper resistance scheme designed to thwart unauthorized tampering of SDR software. Our work is complementary to current software tamper resistance techniques, but, at the same time, is unique in the sense that it is specifically designed for radio software with stringent real-time requirements. To the best of our knowledge, this work represents the first attempt at addressing the important problem of SDR software tampering. The proposed scheme makes an appropriate tradeoff between tamper resistance and execution cost by requiring the program to decrypt the target addresses of selected branch instructions at runtime. The scheme employs a technique called the Random Branch Function Call (RBFC) to enable the programmer to control this tradeoff. Also, we have rigorously evaluated the scheme using various software performance metrics and quantified the relationship between end-to-end delay overhead (caused by the scheme) and voice quality in the context of a voice communication network.

Acknowledgment

This work was supported in part by the National Science Foundation under grants CNS-0627436, CNS-0716208, and CNS-0746925.

References


