N-version Obfuscation

Hui Xu[†‡] Yangfan Zhou[⋆‡]Michael R. Lyu[†‡]
[†]Department of Computer Science, The Chinese University of Hong Kong
[‡] Shenzhen Research Institute, The Chinese University of Hong Kong
[⋆] School of Computer Science, Fudan University

ABSTRACT

Although existing for decades, software tampering attack is still a main threat to systems, such as Android, and cyber physical systems. Many approaches have been proposed to thwart specific procedures of tampering, *e.g.,* obfuscation and self-checksumming. However, none of them can achieve theoretically tamper-proof without the protection of hardware circuits. Rather than proposing new tricks against tampering attacks, we focus on impeding the replication of software tampering via program diversification, and thus pose a scalability barrier against the attacks. Our idea, namely N-version obfuscation (NVO), is to automatically generate and deliver same featured, yet functionally nonequivalent software versions to different machines or users.

In this paper, we investigate such an idea on Android platform. We systematically design a candidate NVO solution for networked apps, which leverages a Message Authentication Code (MAC) mechanism to generate the functionally nonequivalent diversities. Our evaluation result shows that the time required for breaking such a software system increases linearly with respect to the number of software versions. In this way, attackers would suffer great scalability issues, considering that an app can have millions of users, each using different versions. With minimal NVO costs, effective tamper-resistant security can therefore be established.

Keywords

Software Security, Obfuscation, Reverse engineering, Tamper-resistance

1. INTRODUCTION

Software is vulnerable to tampering attacks after release. Attackers may bypass its license checking mechanism to use restricted features, or they may pack malicious payloads into the original software and disseminate repacked versions [53]. More severely, tampering attacks even became a major threat to cyber physical systems (*e.g.,* smartphone, Android auto [1]) as many of them adopt Android as their operating system, which is more vulnerable than conventional embedded systems [26]. Once cyber physical systems are tampered, it may incur not only security issues, but also some safety consequences. To protect software from being tampered, two major classical approaches have been proposed, *i.e.,* obfuscation and self-checksumming. On one hand, software can be protected with obfuscation approaches to deter attackers from interpreting and locating target code spots. On the other hand, software can embed self-checksumming code to detect whether it has been tampered during execution. Nonetheless, once such tricks have been recognized, skillful attackers can design hand-crafted tools to launch attacks. It is often believed that software cannot achieve theoretically tamper-resistance without trusted hardware circuits [10]. But hardware-based approaches suffer compatibility issues with current PC or smartphone taxonomy as they require specifically tailored hardware, hence research on purely software-based approach is nontrivial. In this paper, we investigate a more robust tamper-resistant solution which would not be defeated even due to the exposure of tricks, and does not rely on hardware circuits.

Our approach is based on the fact that besides successfully tampering a software instance, a general tampering attack also includes a replication and dissemination phase, so as to affect more hosts and gain as much benefit as possible. Intuitively, we may not guarantee a software instance to be fully tamper-resistant, but we can nullify the applicability of the tampering solution on general machines, other than the attacker's experimental one. Such an idea is inspired by the existing program diversification approach [19], which prevents the spreading of attacks by making intrusions much harder to replicate. If an attacker wishes to launch the intrusion on another machine, she has to work on it specifically. In this way, we can disarm the ability of automated contagion so as to control the scope of potential damages. According to a recent survey [25], existing software diversification approaches only consider functionally equivalent programs, which can be effective against several kinds of attacks such as return oriented programming [34], but not for tampering. Note that functional equivalence guarantees the universal applicability of a software package on multiple hosts. If a package itself is tampered before installation, its dissemination can hardly be obstructed. Therefore, an effective program diversification

CPSS'16, May 30-June 03, 2016, Xi'an, China.
© 2016 ACM. ISBN 978-1-4503-4288-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2899015.2899026

approach is demanded for impeding software tampering attacks.

As a first attempt, we propose to deliver the same featured, but functionally nonequivalent software versions to different machines. We name the approach as N-version obfuscation, and succinctly describe its major properties: *metamorphic*, *homomorphic*, and *automated*. The *metamorphic* property requires each version of the software to be unique in functionality so as to avoid the infection of tampering; the *homomorphic* property enables a universal handler to handle the difference among versions; and finally, the *automated* property automates the process of the compiling and delivery of the N versions of obfuscated software. We further provide a candidate solution for applications of client-server architecture, *i.e.,* by integrating a MAC mechanism with functionally nonequivalent SHA1 algorithms [3] into the original software, it can be resistant to tampering infection. We show that our candidate solution is applicable to many software integrity protection problems. Finally, it is worth noting that NVO itself provides no protections against tampering; however, it can be applied seamlessly to other existing tamper-resistant approaches, and hence equip them with the replication-resistant property. Our security analysis result shows that the attacking complexity incurred by NVO increases almost linearly with the number of functionally nonequivalent software versions, which would pose a scalability barrier against tampering attacks considering that an application can have millions of versions used by different users.

The rest of this paper is organized as follows. We first give more details about the motivation and background in Section 2. We then introduce our approach with a candidate solution in Section 3. Section 4 evaluates the effectiveness of our approach. The related work is discussed in Section 5. Finally, Section 6 concludes this paper.

2. MOTIVATION AND BACKGROUND

2.1 Adversary Model

In this work, we consider the hostile host model [38], which has been widely adopted by existing software tamper-resistance work, such as [14, 21]. We assume that to launch such tampering attacks, attackers can use malicious host to analyze the software, and they can fully inspect the software execution step by step.

Note that our assumption requires weaker security protection from the host or hardware, and is more general. Other systems (*e.g.,* some cyber physical systems) may have security chips which can provide stronger protection leveraging hardware circuits. Our approach in this paper is also applicable to these scenarios.

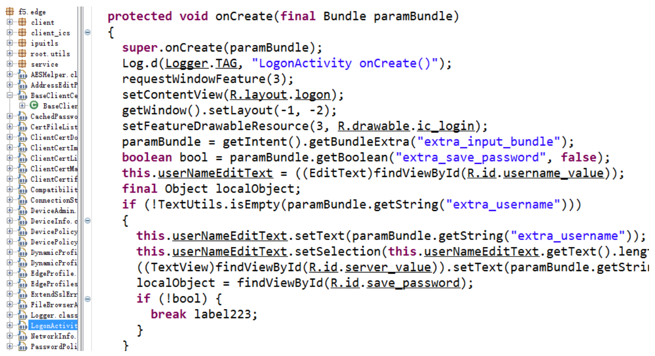
2.2 Software Tampering Insight

Software delivered to end users is vulnerable to tampering. Attackers may modify the original program execution logic for specific purposes. Generally, the modification can be achieved in two ways: software repacking and dynamic injection. We discuss these two approaches as follows.

2.2.1 *Software Repacking*

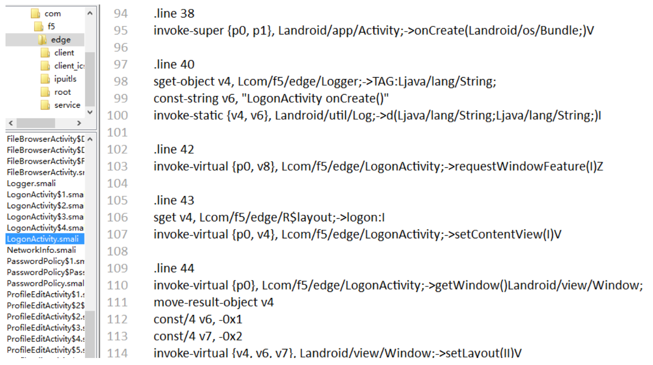
For many reasons, software installation packages downloaded by the users may not be the original ones. Take Android apps as an example, adversaries may replace the original advertisement module within the package for extra profits; or they may have planted malicious payloads in the repacked package. Such repacked Android apps are very common in either Google Play or third-party markets [52].

One important reason for the widespread of Android app repacking is that repacking Android apps is generally much easier than repacking traditional PC software written in C or C++, or iOS apps written in Objective C. Android program is mainly written in Java, and its installation package is delivered to end users in a compressed file (APK) with an `apk` extension. One major component of the installation package is the `classes.dex`, which wraps all the java classes. To interpret the program logic of the app, one may convert the `classes.dex` to either Java bytecode (*i.e.,* `jar` file) or Dalvik bytecode (*i.e.,* smali code) with corresponding tools (*e.g.,* dex2jar, apktool). Unlike the assembly language, the bytecodes are very easy to be interpreted by programmers. Figure 1 shows some examples of such bytecode snippets. The modification of an APK can hence be achieved by rewriting the target bytecode.



```
protected void onCreate(final Bundle paramBundle)
{
  super.onCreate(paramBundle);
  Log.d(Logger.TAG, "LogonActivity onCreate()");
  requestWindowFeature(3);
  setContentView(R.layout.logon);
  getWindow().setLayout(-1, -2);
  setFeatureDrawableResource(3, R.drawable.ic_login);
  paramBundle = getIntent().getBundleExtra("extra_input_bundle");
  boolean bool = paramBundle.getBoolean("extra_save_password", false);
  this.userNameEditText = ((EditText)findViewById(R.id.username_value));
  final Object localObject;
  if (!TextUtils.isEmpty(paramBundle.getString("extra_username")))
  {
    this.userNameEditText.setText(paramBundle.getString("extra_username"));
    this.userNameEditText.setSelection(this.userNameEditText.getText().leng
    ((TextView)findViewById(R.id.server_value)).setText(paramBundle.getStri
    localObject = findViewById(R.id.save_password);
    if (!bool) {
      break label223;
    }
}
```

(a) The Java bytecode browsed in jd-jui



```
.line 38
invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

.line 40
sget-object v4, Lcom/f5/edge/Logger;->TAG:Ljava/lang/String;
const-string v6, "LogonActivity onCreate()"
invoke-static {v4, v6}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

.line 42
invoke-virtual {p0, v8}, Lcom/f5/edge/LogonActivity;->requestWindowFeature(I)Z

.line 43
sget v4, Lcom/f5/edge/R$layout;->logon:I
invoke-virtual {p0, v4}, Lcom/f5/edge/LogonActivity;->setContentView(I)V

.line 44
invoke-virtual {p0}, Lcom/f5/edge/LogonActivity;->getWindow()Landroid/view/Window;
move-result-object v4
const/4 v6, -0x1
const/4 v7, -0x2
invoke-virtual {v4, v6, v7}, Landroid/view/Window;->setLayout(II)V
```

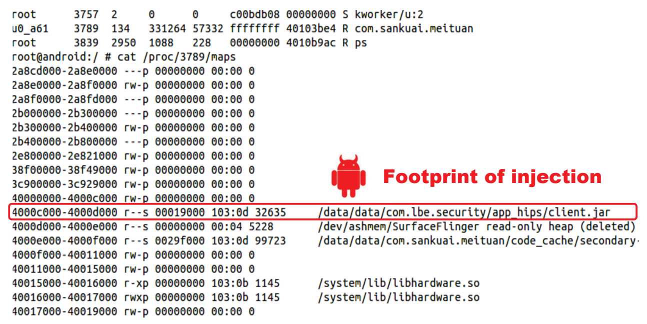
(b) The corresponding smali code

Figure 1: Example code segments disassembled from an Android installation package

It is worth noting that there are already many existing mechanisms to protect users from using repacked apps. Developers may adopt obfuscation techniques during compilation (*e.g.,* using ProGuard [5] offered by Google), which generally hide the meaning of the classes, functions, and variables, by translating their names to meaningless alphabets. Adversaries may encounter much difficulties in understanding the program. However, many important components cannot be obfuscated, such as the standard Android and Java methods. Only with such limited information about the program, adversaries can add payloads, *e.g.,* they may add a payload by simply invoking `system.load()` in `onCreat()`, a function commonly used to start an Android program (or Activity). The discussed obfuscation approaches are mainly effective in protecting the apps from plagiarism, which requires comprehensive understanding about the code. Another official package integrity protection mechanism offered by operating system is a digital signature-based APK originality verification mechanism. However, many personal developers sign the APK with a self-signed digital certificate, which cannot be verified. To promote the convenience for both developers and users, Android generally does not strictly forbid the installation of such APKs signed by untrusted digital certificate, and hence severely degrade effectiveness of the security mechanism.

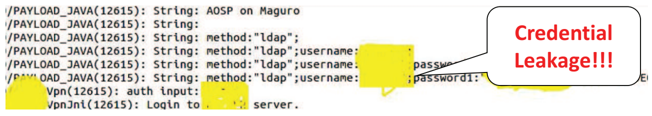
2.2.2 *Dynamic Injection*

Attackers may also manipulate an app during its execution, *e.g.,* by injecting a library into the app process using Linux `ptrace` tool. This approach is widely adopted by viruses, which inject either inspection code to monitor the program execution, or place a back door to control it. Besides viruses, powerful anti-virus software also uses the same way to 'protect' their users' security. Figure 2(a) demonstrates the footprint of dynamic injection launched by a famous Android security software package. The dynamically injected payloads can be very powerful, and cause severe security threats to users. Figure 2(b) demonstrates a credential leakage experiment on a VPN client of a famous vendor (whose name is not disclosed in this paper for security reasons), where we can easily obtain the login credentials by injecting a payload. Moreover, according to another experiment with 100 popular apps, over 90% of them can be dynamically injected [47].



```
root      3757  2     0      0     c00bdb08 00000000 S kworker/u:2
u0_a61    3789  134   331264 57332 ffffffff 40103be4 R com.sankuai.meituan
root      3839  2950  1088   228   00000000 4010b9ac R ps
root@android:/ # cat /proc/3789/maps
2a8cd000-2a8e0000 ---p 00000000 00:00 0
2a8e0000-2a8f0000 rw-p 00000000 00:00 0
2a8f0000-2a8fd000 ---p 00000000 00:00 0
2b000000-2b300000 ---p 00000000 00:00 0
2b300000-2b400000 rw-p 00000000 00:00 0
2b400000-2b800000 ---p 00000000 00:00 0
2e800000-2e821000 rw-p 00000000 00:00 0
38f00000-38f49000 rw-p 00000000 00:00 0
3c900000-3c929000 rw-p 00000000 00:00 0
40000000-4000c000 rw-p 00000000 00:00 0
4000c000-4000d000 r--s 00019000 103:0d 32635      /data/data/com.lbe.security/app_hips/client.jar
4000d000-4000e000 r--s 00000000 00:04 5228       /dev/ashmem/SurfaceFlinger read-only heap (deleted)
4000e000-4000f000 r--s 0029f000 103:0d 99723      /data/data/com.sankuai.meituan/code_cache/secondary·
4000f000-40011000 rw-p 00000000 00:00 0
40011000-40015000 rw-p 00000000 00:00 0
40015000-40016000 r-xp 00000000 103:0b 1145      /system/lib/libhardware.so
40016000-40017000 rwxp 00000000 103:0b 1145      /system/lib/libhardware.so
40017000-40019000 rw-p 00000000 00:00 0
```

(a) The app (pid:3789) has been injected a library by LBE.



```
/PAYLOAD_JAVA(12615): String: AOSP on Maguro
/PAYLOAD_JAVA(12615): String:
/PAYLOAD_JAVA(12615): String: method:"ldap";
/PAYLOAD_JAVA(12615): String: method:"ldap";username:[redacted]
/PAYLOAD_JAVA(12615): String: method:"ldap";username:[redacted]passw
/PAYLOAD_JAVA(12615): String: method:"ldap";username:[redacted];password1:"[redacted]E
[redacted]Vpn(12615): auth input:[redacted]
VpnJni(12615): Login to [redacted] server.
```

(b) A credential hacking experiment by dynamically hijacking the function `java...tostring()`.

Figure 2: Examples of dynamic injection

2.3 Tamper-resistance Background

In this section, we first overview the reverse engineering techniques, and then discuss several major approaches against specific procedures of reverse engineering.

2.3.1 *Reverse Engineering Overview*

Reverse engineering is the core technology for software tampering. It involves a process that analyzes and manipulates a software package based on its executables, *e.g.,* in an executable and linkable format (ELF). Anti-reversing techniques impede such a process by placing tricks in the executables to fool the analyzer. General reverse engineering on ELF files involves two phases: a disassembly phase and an analyzing phase. The disassembly phase decodes the ELF binaries to assembly code, which can be performed automatically by some tools (*e.g.,* IDA). We can hardly impede the decoding because eventually the processor has to be able to decode and execute the file. Therefore, the reverse engineering and corresponding anti-reversing efforts mainly lie in the analysing phase.

There are two general ways to do reverse analysis, *i.e.,* the offline approach and the online approach. The offline approach does not execute the assembly code, but directly analyzes it using static reverse engineering tools such as IDA [4]. If a program has not been properly obfuscated, its control flow graph (CFG) can be easily derived, which shows the assembly code in blocks, and indicates their call relationships. In this way, the complexity of analyzing the assembly code can be simplified. CFG can provide great assistance for reverse engineers to grasp the meaning of the low-level assembly code which has little semantics. Even though many powerful offline analysis tools are available off-the-shelf, pure offline analysis still suffers hard limitations in detecting some anti-reversing protections, *e.g.,* runtime code unpacking is widely used by malware to escape static analysis. Therefore, adversaries may also execute the code to obtain execution instruction traces [35] or to debug the code step by step and perform the analysis, which is known as online reverse-engineering [17]. Such an analysis process is generally not affected much by obfuscation [49], and adversaries can leverage a set of system monitoring tools to monitor the output of a code snippet, which facilitates the reverse engineering process.

There are many tricks that have been proposed to obstruct the analyzing phase. Next, we discuss the major approaches and their limitations.

2.3.2 *Obfuscation*

To protect programs from being analyzed, a few obfuscation approaches have been proposed [50]. The main idea of obfuscation is to introduce some redundancies into the original program, while the original functionality of the program is still preserved. Several methods have been proposed to achieve this. For example, a developer can use opaque constant to add blocks of junk code that would never be executed. Fig. 3(a) shows such an example. The developer may further create NP-hard problems by introducing pointer analysis difficulties as discussed in [24] and [32]. Another method is to confuse the trigger condition of one code block with one-way function, so that the static analyzer cannot infer whether the block would be executed or is redundant. Fig. 3(c) shows an example of transferring an obvious condition into an opaque condition with a hash function. Besides, unsolved conjectures have been proposed to confuse the exit criteria of loops, *e.g.,* Fig. 3(b) is an example of using Collatz Conjecture. Such obfuscation techniques can introduce great difficulties to general static analysis tools for analyzing the CFG and grasping the meaning of assembly code. However, all the obfuscation approaches have vulnerabilities. For example, opaque constant is vulnerable to symbolic execution which implements a smart constraint solver, and unsolved conjectures are vulnerable to homemade tools which can recognize the patterns of conjectures [30, 48]. Barak *et al.* have theoretically proved black-box obfuscation is not possible [7]. We conclude the limitation of obfuscation techniques as Collberg stated in [13]:

> "Given enough time, effort and determination, a competent programmer will always be able to reverse engineer any application."

```
int b = getchar();
//always true
if ( $7a^2 - 1 \neq b^2$ ) {
     foo();
}
else {
     junk();
}
```

(a) Opaque constant

```
int x = 2000;
while ( x > 1 ) {
   if ( x % 2 == 1 ) {
      x = 3 * x + 1;
   }
   else   x = x / 2;
   if ( x == 1)
      foo();
}
```

(b) Collatz Conjecture

```
/*Original code:*/
if (  b == 1 )
    foo();
```

```
/*After obfuscation:*/
if ( hash(b) == hashvalue )
    foo();
```

(c) One way function

Figure 3: Demonstration of obfuscation approaches with different tricks. The original code for Fig. 3(a) and Fig. 3(b) is *foo();*

2.3.3 *Anti-debugging*

Researchers have suggested to set traps with anti-debugging code to hinder debugging. For example, one may simply check the debug register to detect if a debugger is present, or count the execution time of a code block to detect if it has been paused, and then penalize the debugger [20, 41]. Again, if the trick of anti-debugging code is recognized, adversaries can suppress the checking by patching the binaries, or simply switching to another debugger.

2.3.4 *Self-checksumming*

When deriving enough understanding about the code, adversaries can manipulate the binaries by adding or deleting some code according to a specific purpose while preserving its ability of execution. A possible way to detect such code patching is to use self-checksumming code. The basic idea is to pre-calculate a value of relative address (*i.e.,* the checksum), and let the program fetch instructions during execution according to such a value. If the checksum governed regions have been tampered, the instruction would not be correct, and the program would likely to suspend [46]. Using overlapped self-checksumming code can further increase the strength of protection. However, it can be defeated by carefully detecting and removing them [36] or exploring the vulnerabilities [46] of the execution environment.

2.3.5 *Watermarking*

Watermarking is an approach for software plagiarism detection. General watermarking approaches embed stealthy watermark messages into the software to declare the ownership, which can be extracted and verified by arbitrators who know the secrets to extract the watermark. Due to security reasons, traditional watermark verification is generally conducted offline only by a few organizations. However, they cannot meet the requirement for detecting fake apps. Recently, several new watermarking approaches have been proposed for Android apps, which can be used for online watermark verification and detecting repacked apps accordingly, such as [37, 52]. Although helpful in code originality verification, it cannot detect whether a software package has been tampered, if the original authorship information (*e.g.,* package name) has been kept. In other words, watermarking cannot prevent software tampering but instead it discourages software piracy.

To conclude, there is still no overwhelming anti-reverse engineering method, *i.e.,* software can never be made fully resistant to tampering without hardware protection.

2.4 Challenge of Tamper-resistant Apps

General applications can be very complex. They can involve classes written in Java, native code written in C or C++, and other third-party libraries, all of which are vulnerable to be tampered. Therefore, a universal safeguard is required to protect the integrity of each component. Since no general tools can be applied for such heterogeneous code, the implementation of tamper-resistance to the whole program would be labour intensive. Moreover, mobile apps are usually upgraded more frequently than general PC software, so that their testing strategy and releasing criteria cannot be as rigorous as PC. Consequently, the labourious tamper-resistant implementation and testing would likely to slow down the releasing speed, or insufficient testing on such low-level code tends to cause more bugs.

Besides applicability issues with respect to software development lifecycle, overhead should also be carefully considered. Traditional anti-tampering approaches usually work by adding extra code to the original program, which can complicate the control flow of the original program, or by performing some integrity checking. Such approaches inevitably incur overhead, and a trade-off between the effectiveness and the overhead should be considered. In order to run smoothly on a resource-constrained mobile device, the app cannot afford much overhead.

Finally, according to the literature [10], it is impossible for software to be absolutely secure against analysis without specific hardware protection. Although there are some existing solutions for Android apps, such as the ProGuard offered by Google [5], DexGuard [2], and AppInk [52], there is no general criteria about what tamper-resistant level can be acceptable, *i.e.,* secure enough. In this regard, app developers would face difficulties in choosing an anti-tampering approach among various approaches off-the-shelf. It is urgent to develop an anti-tampering solution whose effectiveness can be quantified, and referenced by the developers.

3. OUR PROPOSED APPROACH

While achieving theoretically tamper-proof is hardly possible, our idea is to pose the tampering attack unscalable

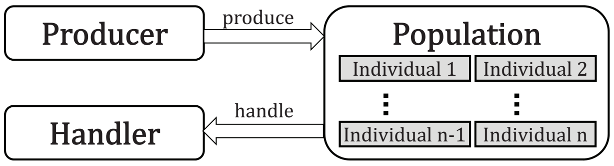


Figure 4: The conceptual framework of NVO

with an NVO approach. In this section, we formally define the idea of NVO, and then discuss a candidate solution with its application scenario.

3.1 General Idea of NVO

We formally define the NVO problem as the following: Given an algorithm A, how to automatically generate a large set of functionally nonequivalent algorithms $\{C_1, ...C_n\}$, which are similar to A, and their parent algorithm P, so that they meet the following three properties:

Homomorphic: When performing on the same task, P can output the same result as C_i, if the gene vector $\{g_1, ...g_n\}$ of C_i is known to P.

Metamorphic: When performing on the same task, C_i and C_j generally output different results.

Automated: The generation and delivery of $\{C_1, ...C_n\}$ can be automated.

Figure 4 demonstrates the conceptual framework of NVO. The producer generates a set of functionally nonequivalent individuals, *i.e.,* $\{C_1, ...C_n\}$. To communicate with each individual and process their requests, the handler is equipped with the parent algorithm P. Suppose the software architecture is in client-server mode, we can deploy the handler at the server side, and deliver the individuals to the client side. In this way, the client can have functionally nonequivalent diversities according to the metamorphic property, and the homomorphic property enables the server to handle such diversities. To make the idea practical, the generation and delivery of such diverse software versions should be automated.

3.2 A Candidate Solution

To apply NVO on apps, one major issue to address is regarding which part of an app can have effective functionally nonequivalent diversities. Intuitively, there are two possible ways: we can either find the candidate code snippet in the original program, or add some extra code to the original program. Generally, the first approach is program dependent, and can hardly be generalized. Hence, our approach is to add extra code which can achieve the intended diversities.

A possible way is to add MAC to the original program. MAC is a popular mechanism adopted by client-server computing architecture to check the integrity and authenticity of messages. When a client sends a request to the server, it calculates the MAC of the request and appends it to the original request. The server validates the MAC first and then processes the request. We can leverage the MAC to create clients with functionally nonequivalent diversities. More specifically, the diversity can be introduced based on the hash algorithm (*e.g.,* the Secure Hash Algorithm 1 (SHA1)), which is one major component of a MAC algorithm. Fig. 5 illustrates such a mechanism. Each client

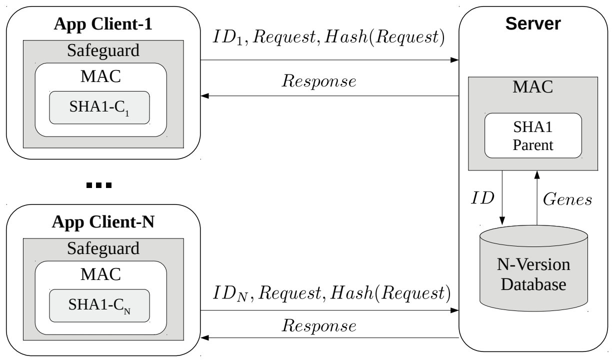


Figure 5: A sample application of NVO for tamper resistant apps

is embedded with a unique SHA1-based MAC calculation algorithm. To successfully perform a request to the server, it has to send the identification (such as machine serial number or user id), the request, and the MAC together to the server. The server queries the genes of a client from its local N-version database according to the identification of the client, and then verifies the MAC. The distribution of such diverse programs can be achieved by implementing the MAC in mobile code (*i.e.,* a dynamic library), and delivering it by the server upon request. In other words, the client software can be launched without the library at the first time and then requests the server for the library. The server randomly chooses a library from a pool of pre-compiled libraries and delivers it to the client; in the meanwhile, the server records the mapping between the genes of the client and its unique identification in the N-version obfuscation database. A detailed safeguard delivery and initialization process is shown in Figure 6.

In the following paragraphs, we first show a viable means to solve the NVO problem with SHA1 algorithm, and then discuss the security measurements which can be built on the mechanism.

3.2.1 N-version Obfuscated SHA1

Our approach leverages the iterations of calculations needed by SHA1 to generate functionally nonequivalent diversities. The main loop of original SHA1 (Algorithm 1) includes 80 rounds of iterations. Each iteration takes one plaintext block ($w[i]$) into calculation. For every twenty rounds, the calculation (the equation for generating f and the value of k) switches to another one. Even though there are some security considerations of choosing a specific calculation for each round, to our best knowledge, no evidence shows the programs would suffer great security degradation if we switch them with each other. Therefore, we can diversify the original SHA1 algorithm by choosing different sequences of equations for generating f and sequences of values of k, which are the genes of individuals. We can also design a parent algorithm which can receive the genes of an individual, and process data input according to the setting of genes. Algorithm 2 shows such a parent algorithm we designed. In Algorithm 2, the pointer array of equations ($f\_genes[80]$) for generating f and the value array of k ($k\_genes[80]$) for the 80 rounds of iterations are passed to the algorithm as the genes of a child. It is clearly seen that, given

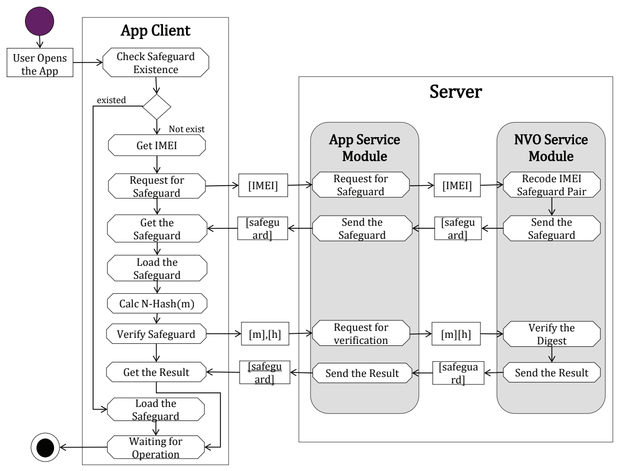


Figure 6: A sample activity diagram for automating safeguard delivery and initialization. The diagram uses IMEI as the unique id that maps with the safeguard version. [m,h] is the request message and the corresponding hash value.

the same input $w[80]$, the parent algorithm can compute the same result as a child when $f\_genes[80]$ and $k\_genes[80]$ are properly set.

```
Data: $w[80]$
// blocks of plaintext
for $i = 0; i < 80; i++$ do
    if $0 \le i \le 19$ then
        $f \leftarrow (b$ AND $c)$ OR ((NOT $b)$ AND $d)$;
        $k \leftarrow$ 0X5A827999;
    end
    if $20 \le i \le 39$ then
        $f \leftarrow b$ XOR $c$ XOR $d$;
        $k \leftarrow$ 0X6ED9EBA1;
    end
    if $40 \le i \le 59$ then
        $f \leftarrow (b$ AND $c)$ OR $(b$ AND $d)$ OR $(c$ AND $d)$;
        $k \leftarrow$ 0X8F1BBCDC;
    end
    if $60 \le i \le 79$ then
        $f \leftarrow b$ XOR $c$ XOR $d$;
        $k \leftarrow$ 0XCA62C1D6;
    end
    $temp \leftarrow (a$ LEFTROTATE 5$) + f + e + k + w[i]$;
    $e \leftarrow d$;
    $d \leftarrow c$;
    $c \leftarrow b$ LEFTROTATE 30 ;
    $b \leftarrow a$;
    $a \leftarrow temp$;
end
```

Algorithm 1: The main loop of SHA1

3.2.2 Security Built-on MAC

The N-version obfuscated SHA1 program itself provides little effectiveness against software tampering attack. However, it is resistant to replication, because the MAC of replicated programs cannot be verified by the server. Therefore, it can serve as a basis for software integrity checking, and equip programs with replication-resistance property. In this section, we discuss one possible way to build such security features on top of the MAC.

```
Data: $f\_genes[80], k\_genes[80], w[80]$
for $i = 0; i < 80; i++$ do
    Call $f\_genes[i]$;
    // Pointer to F0, F1, F2 or F3
    F_TAIL($k\_genes[i], w[i]$);
end
Function F0()
    $f \leftarrow (b$ AND $c)$ OR ((NOT $b)$ AND $d)$;
Function F1()
    $f \leftarrow b$ XOR $c$ XOR $d$;
Function F2()
    $f \leftarrow (b$ AND $c)$ OR $(b$ AND $d)$ OR $(c$ AND $d)$;
Function F3()
    $f \leftarrow b$ XOR $c$ XOR $d$;
Function F_TAIL($k, w$)
    $temp \leftarrow (a$ LEFTROTATE 5$) + f + e + k + w$;
    $e \leftarrow d$;
    $d \leftarrow c$;
    $c \leftarrow b$ LEFTROTATE 30;
    $b \leftarrow a$;
    $a \leftarrow temp$;
```

Algorithm 2: A parent algorithm for SHA1

```
Data: $dict < segment >$
// A list of predefined segment with name and size
Function IntegrityChk()
$pid \leftarrow$ getpid();
$file \leftarrow$ open (/proc/pid/maps);
while $line \leftarrow readline(file)$ != $EOF$ do
    $segName \leftarrow$ GetSegName($line$);
    $segSize \leftarrow$ GetSegSize($line$);
    if !$dict.contains(segNmae)$ then
        Reaction();
    else
        if $dict.getsize(segNmae)$!=$segSize$ then
            Reaction();
        end
    end
end
```

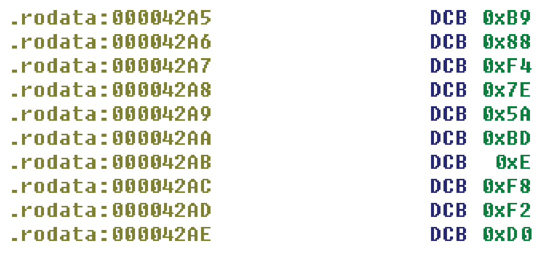
Algorithm 3: An exemplary integrity checking function

A viable means is to implement an integrity checking function aligning with the MAC in the safeguard, so that it can serve as a safeguard for the whole app. By interleaving the code of the integrity checking function with the MAC algorithm, the integrity checking can be triggered when calculating a MAC. Algorithm 3 shows an exemplary integrity checking function for the apps of Android operating system. The function navigates the maps file of the app process itself, which records the program segments and their addresses in the memory. It then compares the record with a previously defined standard dictionary by the developers. If there is any abnormal segment in the maps, *i.e.,* the integrity has been violated, a responsive mechanism can be triggered. Such an approach is effective in detecting either software repacking or dynamic injection attacks as we have discussed in Section 2. For example, Algorithm 3 can detect the tampering in Fig. 2(a) by finding that `com.lbe.../client.jar` is an abnormal segment.

If an attacker has successfully tampered one copy of the safeguard (*e.g.,* removing the integrity checking function) and replicated it on other machines, the server can detect the replication because of an incorrect MAC, *i.e.,* inconsistent mapping between the identification and the genes. We may further implement a reaction mechanism to renew the safeguard or crash the client software directly.

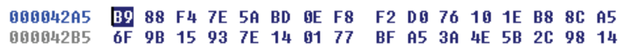
3.2.3 *Protect the Genes*

Genes are the secrets for the diversity, and should not be easily extracted by adversaries. Without protection, the N-version software executables are generated in plain ELF binaries. Adversaries may find the gene sections by comparing several versions of the software, and extract the genes manually, or even automatically. Figure 7 demonstrates the genes of the safeguard located using IDA. To provide protections for the secrets from being extracted, we randomly change the meaning of the genes, *i.e.,* the same value of $f\_genes[i]$ for different versions may trigger different operations. We further adopt two methods to protect the meaning of genes from being reasoned: functional obfuscation, and control flow obfuscation with opaque constant.



```
.rodata:000042A5                    DCB 0xB9
.rodata:000042A6                    DCB 0x88
.rodata:000042A7                    DCB 0xF4
.rodata:000042A8                    DCB 0x7E
.rodata:000042A9                    DCB 0x5A
.rodata:000042AA                    DCB 0xBD
.rodata:000042AB                    DCB  0xE
.rodata:000042AC                    DCB 0xF8
.rodata:000042AD                    DCB 0xF2
.rodata:000042AE                    DCB 0xD0
```

(a) Genes in the .rodata section of the safeguard using IDA View



```
000042A5  B9 88 F4 7E 5A BD 0E F8  F2 D0 76 10 1E B8 8C A5
000042B5  6F 9B 15 93 7E 14 01 77  BF A5 3A 4E 5B 2C 98 14
```

(b) The bits of genes using Hex View

Figure 7: The genes can be located with IDA

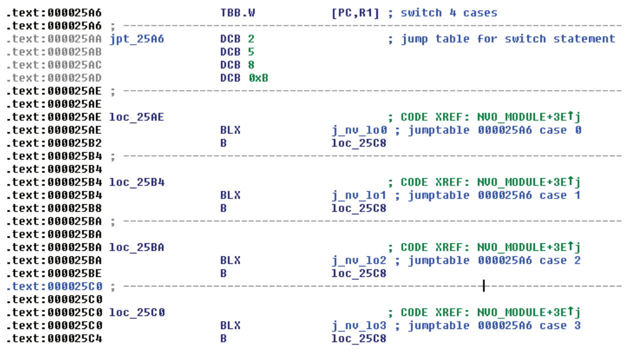
a) *Functional obfuscation*: Adversaries may reason the meaning of the gene by checking the call relationship with some functions (*e.g.,* $F0, F1$ in Algorithm 2). We hence obfuscate the functions in each version from being located. Firstly, we change the function names to random strings, so that the functions can be easily located by their names. Secondly, we change the order of those functions, so that they appear at different positions of the executables. In this way, even the genes are extracted, adversaries will still have trouble to map the genes with the functions.

b) *CFG obfuscation*: In this step, we obfuscate the CFG, so that even the functional obfuscation can be penetrated, the calling relationship between the genes and the functions would not be easily solved. To this end, we adopt the obfuscation approaches proposed in [32], which composes NP-hard problems with function pointers and opaque constants. A comparison of the instructions before and after the CFG obfuscation is shown in Figure 3.2.3.

Finally, it is worth noting that the obfuscation protections we adopt to protect the secrets in this section are all functional equivalent transformations. The NVO approach itself does not provide any resistance to reverse engineering. However, our approach can be seamlessly integrated with other aiti-reverse engineering protections, such as anti-checksumming. We may use them together to provide better tamper-resistant capabilities.

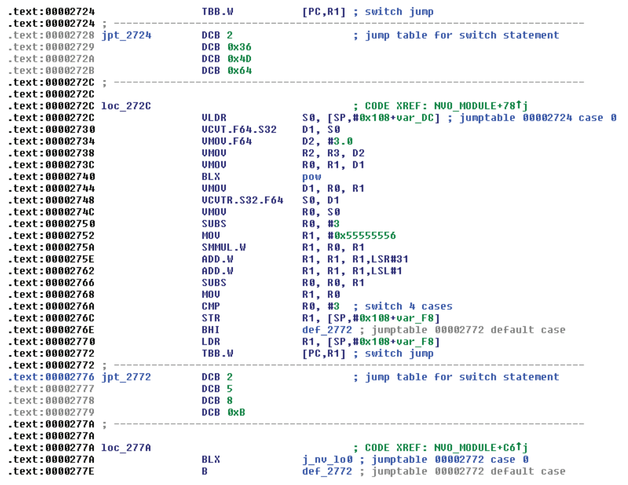
3.2.4 *Automated N-version Generation*

We automate the process of generating N-version SHA1 algorithms based on LLVM, which is a widely used open-



```
.text:000025A6                 TBB.W           [PC,R1] ; switch 4 cases
.text:000025A6 ; ---------------------------------------------------------------------------
.text:000025AA jpt_25A6        DCB 2                   ; jump table for switch statement
.text:000025AB                 DCB 5
.text:000025AC                 DCB 8
.text:000025AD                 DCB 0xB
.text:000025AE ; ---------------------------------------------------------------------------
.text:000025AE
.text:000025AE loc_25AE                                ; CODE XREF: NVO_MODULE+3E↑j
.text:000025AE                 BLX             j_nv_lo0 ; jumptable 000025A6 case 0
.text:000025B2                 B               loc_25C8
.text:000025B4 ; ---------------------------------------------------------------------------
.text:000025B4
.text:000025B4 loc_25B4                                ; CODE XREF: NVO_MODULE+3E↑j
.text:000025B4                 BLX             j_nv_lo1 ; jumptable 000025A6 case 1
.text:000025B8                 B               loc_25C8
.text:000025BA ; ---------------------------------------------------------------------------
.text:000025BA
.text:000025BA loc_25BA                                ; CODE XREF: NVO_MODULE+3E↑j
.text:000025BA                 BLX             j_nv_lo2 ; jumptable 000025A6 case 2
.text:000025BE                 B               loc_25C8
.text:000025C0 ; ---------------------------------------------------------------------------
.text:000025C0
.text:000025C0 loc_25C0                                ; CODE XREF: NVO_MODULE+3E↑j
.text:000025C0                 BLX             j_nv_lo3 ; jumptable 000025A6 case 3
.text:000025C4                 B               loc_25C8
```

(a) Before CFG obfuscation, the function calling can be easily mapped with the genes in the jump table.



```
.text:00002724                 TBB.W           [PC,R1] ; switch jump
.text:00002724 ; ---------------------------------------------------------------------------
.text:00002728 jpt_2724        DCB 2                   ; jump table for switch statement
.text:00002729                 DCB 0x36
.text:0000272A                 DCB 0x4D
.text:0000272B                 DCB 0x64
.text:0000272C ; ---------------------------------------------------------------------------
.text:0000272C
.text:0000272C loc_272C                                ; CODE XREF: NVO_MODULE+78↑j
.text:0000272C                 VLDR            S0, [SP,#0x108+var_DC] ; jumptable 00002724 case 0
.text:00002730                 VCVT.F64.S32    D1, S0
.text:00002734                 VMOV.F64        D2, #3.0
.text:00002738                 VMOV            R2, R3, D2
.text:0000273C                 VMOV            R0, R1, D1
.text:00002740                 BLX             pow
.text:00002744                 VMOV            D1, R0, R1
.text:00002748                 VCVTR.S32.F64   S0, D1
.text:0000274C                 VMOV            R0, S0
.text:00002750                 SUBS            R0, #3
.text:00002752                 MOV             R1, #0x55555556
.text:0000275A                 SMMUL.W         R1, R0, R1
.text:0000275E                 ADD.W           R1, R1, R1,LSR#31
.text:00002762                 ADD.W           R1, R1, R1,LSL#1
.text:00002766                 SUBS            R0, R0, R1
.text:00002768                 MOV             R1, R0
.text:0000276A                 CMP             R0, #3  ; switch 4 cases
.text:0000276C                 STR             R1, [SP,#0x108+var_F8]
.text:0000276E                 BHI             def_2772 ; jumptable 00002772 default case
.text:00002770                 LDR             R1, [SP,#0x108+var_F8]
.text:00002772                 TBB.W           [PC,R1] ; switch jump
.text:00002772 ; ---------------------------------------------------------------------------
.text:00002776 jpt_2772        DCB 2                   ; jump table for switch statement
.text:00002777                 DCB 5
.text:00002778                 DCB 8
.text:00002779                 DCB 0xB
.text:0000277A ; ---------------------------------------------------------------------------
.text:0000277A
.text:0000277A loc_277A                                ; CODE XREF: NVO_MODULE+C6↑j
.text:0000277A                 BLX             j_nv_lo0 ; jumptable 00002772 case 0
.text:0000277E                 B               def_2772 ; jumptable 00002772 default case
```

(b) After CFG obfuscation, the function calling related to the genes has been obfuscated using opaque constants and sub jump tables.

Figure 8: A control flow obfuscation example for the switch-case code block

source compiler that supports extensions. LLVM first represents the source code with Abstract Syntax Tree (AST), and then transfers it into intermediate code (IR), which would finally be compiled into executables according to a specific platform. The automation can be achieved in two ways: in AST level by customizing a `libtooling` (*i.e.,* a LLVM tool that can manipulate the source code of a target AST branch during the compilation process), or in IR level by adding extra N-version obfuscation passes to the compiler. We suggest the second way because IR is language and machine independent, and thus more advantageous in crafting an obfuscator with better adaptability.

According to Algorithm 2, each gene (either $fp[i]$ or $k[i]$) has four possibilities, so we can use two bits to represent a gene. In each compilation, we first randomly generate two 160-bit long sequences: one as the chromosome for the equation function pointer (*i.e.,* $fp[80]$) and the other as the chromosome for the value option of k (*i.e.,* $k[80]$). We then replace the corresponding code with hardcoded genes. Similarly, we can implement the obfuscation approaches by adding obfuscation passes for protecting the genes in a similar way as that in [22].

3.3 Approach Discussion

Several ideas proposed in literatures are very close to NVO, such as white-box encryption, and N-version programming (NVP). In this section, we compare NVO with these ideas to show their difference and clarify why NVO is a unique approach for security.

3.3.1 White-box Encryption

NVO creates functional nonequivalent diversities among versions in the level of program logics. A question to ask is why we do not simply use different keys to compose diversities? For example, we may use a keyed-hash message authentication code (HMAC) algorithm and hardcode a unique symmetric key into each version. Note that such an approach is also effective, but it is more vulnerable than our proposed NVO approach, because hiding a key (*i.e.,* white-box cryptography) is more difficult than hiding the program logic [11]. White-box cryptography can be viewed as an extreme circumstance of NVO with only key diversities. Besides, white-box cryptography does not stress on producing diversities, which is the major focus of NVO. Essentially these two approaches are two orthogonal frameworks, each with its own objectives and algorithms. Nevertheless, our approach may incorporate white-box cryptography for a hybrid security mechanism.

3.3.2 N-version Programming

NVO improves software security by automatically generating different versions of software. The idea is inspired by the classical N-version Programming (NVP) approach, which improves software reliability by independently designing different versions of software, so that the same bug may not happen in all versions [9, 29]. Although the two ideas are similar, they target on solving different issues, and they are very different in several key aspects. A detailed comparison of NVO versus NVP is show in Table 1.

Table 1: A comparison of NVP versus NVO

	NVO	NVP
Purpose	Security: tampering resistant	Reliability: fault tolerant
Fault	Malicious faults	Accidental faults
Assumption	Independent obfuscation	Independent programming
Program	Functionally nonequivalent	Functionally equivalent
Generation	Automatically generated	Independently designed
Population	Very large	Very small
Effectiveness	$O(N)$ security	$1-(1-R)^N$ reliability
Cost	$O(1)$	$O(N)$

4. EFFECTIVENESS EVALUATION

The goal of our work is to impede tampering replication by creating diverse software instances, and thus increasing the tampering complexity for intruding multiple clients. In this evaluation section, we first discuss the effectiveness of NVO in thwarting tampering replication, and then evaluate the complexity incurred by NVO for intruding multiple software clients. Our evaluation process only considers the software tampering attack, and does not consider other types of attacks, such as side-channel attacks, or attacks in the network layer.

4.1 Approach Effectiveness

Suppose a program has adopted the protection mechanism discussed in Section 3.2. A decent attacker wishes to manipulate the program binaries for a specific purpose, through either software repacking or dynamic injection. To this end, she has to disarm the security safeguard by removing or modifying the security code discussed in Algorithm 3. According to the adversary analysis, the safeguard cannot be simply removed or disabled from the app, because the MAC mechanism rested in the safeguard needs to be executed. However, in a hostile host environment, the software can be fully inspected. Through careful analysis, the attacker may discern that the protection lies in the integrity checking function of Algorithm 3. If she is skillful and spends enough efforts, she can further disable the checking by carefully modifying the function, such as suppressing the reaction. If there is no NVO protection, the attacker may replicate the repacked app, or apply her dynamic injection scripts on other machines, and the whole software system would be contaminated. However, NVO can impede such replication of tampering attack, with detailed discussions in what follows.

If the attacking type is app repacking, then the repacked app replicated on multiple machines would have the same genes for the MAC algorithm. Suppose the app (*e.g.,* ebank) uses `UserID` as the corresponding unique `ID` for the genes (as we have discussed in Figure 5), then the server would receive mismatching MAC from the app that has been logged on, and thus can detect that the client app has been tampered. In this way, we may take advantage of the user's credential, which cannot be easily faked. But what if the app mainly provides services to anonymous users that do not require logging on? Generally, such kind of apps do not have strong security requirement. Having said that, NVO still works for such apps by employing other information as the `ID`, such as the International Mobile Equipment Identity (IMEI). The major difference is that IMEI can be faked much easier. For example, the repacked app can hardcode the faked IMEI corresponding to the genes of the app. However, when replicating on multiple machines, the server would detect the abnormality that multiple clients are using the same IMEI. In a nutshell, due to the divergence property of NVO, the server can detect app repacking attack when the app is communicating with the server. Such a detection condition is trivial because for many apps, pure clients are useless unless they can interact with the server (*e.g.,* ebank, shopping), or obtain rich contents from the server (*e.g.,* news, videos).

If the attacking type is dynamic injection, the sample integrity checking function (*e.g.,* Algorithm 3) is effective in detecting the tampering. Although it relies little on the NVO mechanism, the NVO hardens the security of the integrity checking function against being suppressed. For example, if the attacker seeks to suppress the security checking in Algorithm 3, an intuitive way is to disable the `Reaction()` function. To locate the function within the ELF file, the attacker may either check the related ELF table (*e.g.,* .dynsym and .rel.plt) dynamically or hardcode the address of the function into the malicious code. However, our NVO implementation transforms such selfdefined function names to a random alphabet combination for each version, so that the dynamic approach cannot know which symbol designates the target function. Besides, the hardcoded address also cannot work because the function would appear at different positions of the binaries for each version, due to our functional obfuscation implementation.

4.2 Protection Strength

To enable the replication of tampering attacks, the attacker need to bypass or thwart our NVO settings. Intuitively, she may either suppress the security checking in each version dynamically or she may create a library which is similar to the parent algorithm, and extracts the genes of each version dynamically. We discuss the complexity for these two kinds of attacks in what follows.

4.2.1 Disarm Security Checking

In order to suppress the security checking one machine, the attacker has to obtain the safeguard on that machine, and then remove the checking instructions within the safeguard. If the safeguard is protected with interleaved self-checksumming code [8], a successful tampering requires removing all the self-checksumming code at the same time, of which the chance is very low without sophisticated analysis. Existing approaches on identifying such code generally require dynamic taint analysis and debugging [36]. Empirically, the time required to tamper each safeguard is not negligible.

Let t_0 denote the time needed for analyzing one software copy and tampering it on the attacker's own hostile host. The time complexity is $O(1)$, which equals to tampering one software copy without NVO. Let t_1 denote the time needed to fetch the safeguard on another machine, so as to replace it, and t_2 denote the time needed to tamper it. If the attacker wishes to tamper the software on n machines, the total time can be estimated as $t_0 + n * (t_1 + t_2)$. Because of the interleaved self-checksumming code, t_2 should not be negligible [35], hence the complexity can be approximated to $O(n)$.

4.2.2 Universal Attacker

Another possible tampering approach is to build an algorithm similar to the parent algorithm, which calculates the hash value according to the genes of a specific child. Such an approach requires the attacker to be able to extract the genes from each safeguard. To this end, the attacker may compare the difference between two implementations, and locate the genes. If the attacker has derived enough knowledge on our NVO theory and implementation, such kind of attack is theoretically possible. However, in our NVO implementation, the meanings of the genes differ in different versions, because they have been obfuscated with opaque constants [32]. To our best knowledge, existing work on breaking such obfuscated programs requires either symbolic execution with sophisticated constraint solvers or complicated taint analysis [49], which is computational intensive and time-consuming. Let t_3 denote the time needed to extract the genes of a safeguard. The time needed to tamper the software system can be estimated to $t_0 + n * (t_1 + t_3)$. Because efficient automatic deobfuscation is hard to achieve, t_3 should not be negligible [6, 16]. Therefore, the complexity still equals to $O(n)$. Note that n can be made arbitrarily large as the obfuscation task can be fully automated.

Finally, our complexity analysis results rely on the problem incurred by traditional anti-tampering protections. But different from the traditional work, we do not require the anti-tampering protections to approach theoretical secure, which can hardly be guaranteed. We only require that the protections cannot be thwarted automatically, which is more sound and realistic.

5. RELATED WORK

In this section, we first overview the literature about Android app tampering and protection, which takes advantage of a specific platform; we then review important work in software reverse engineering area, which is more general. Finally, we discuss the work that applies program diversification to improve software security and reliability, and discuss our difference with them.

5.1 Tamper-resistant Apps

To protect apps from unauthorized manipulation, Google offers ProGuard [5], which is a free Android program obfuscater and optimizer that can make the application harder to be analyzed. Comparatively, DexGuard [2] is a commercial tool build on ProGuard, and is more powerful. However, they both provide no features against tampering replication.

Recent literature in protecting users from using tampered apps mainly focuses on detecting repacked apps in a large scale, such as [18, 39, 43, 44, 53, 54]. However, these investigations are not quite related to our problem. Several other investigations focus on detecting repacked apps with watermarking approaches [37, 52]. Wu *et al.* propose the idea of manifest app and the corresponding tool, AppInk [52]. AppInk can take the source code of an app as input and automatically generate a new app with a transparently-embedded watermark and the associated manifest app. The manifest app can then be used for verification purpose by triggering certain app control flows to regenerate the watermark. Ren *et al.* propose another watermarking solution (*i.e.,* Droidmarking) for app plagiarism detection. Droidmarking is based on a primitive called self-decrypting code, and the watermark locations are not intentionally concealed. These watermarking approaches are generally effective for app repacking detection within the scope of code plagiarism, but they are not effective for other kinds repacking, such as third-party library replacement.

To protect the apps from being repacked, Wu *et al.* propose an approach that re-encodes an Android app with a transformed virtual instruction set, so that the app cannot be inspected by general reverse engineering tools [51]. To run the protected app, the developers can use a specialized execute engine for these virtual instructions. The idea is similar to another work proposed by Shu *et al.* [42]. These approaches can increase difficulties for interpreting the program, and are effective against popular reverse engineering tools. But their security relies on the secret of the instruction translation table, which can be discovered manually by decent adversaries.

As we have discussed, existing work in this area mainly focuses on increasing the difficulty of repacking, or the detection of repacked apps. Our work is different from them in that we focus on decreasing the reusability of repacked app.

5.2 Reverse Engineering

Software protection is a research problem since decades ago. The proposed solutions are generally two-fold: hardware circuit assisted solutions which provide better security assurance, or pure software solutions which enjoy better adaptability than general hardware [10]. For our research problem, hardware circuit assisted solutions are not applicable because of their requirement on specific hardware, so we mainly discuss the pure software solutions.

The literature on software protection with anti-reverse engineering approaches aims at different purposes. While some researchers look for protections against piracy [27, 32] and intrusion [8], others investigate on impeding malware against detection [31, 33, 40]. However, they share a set of common protection techniques with only slight difference. Obfuscation is a basic software protection approach. It can complicate the binaries, and increases the difficulty of the reverse engineering. Ogiso *et al.* propose to obfuscate the code by constructing an NP-Hard complexity problem, which requires to determine the real function pointer from an array of pointers [32]. However, this approach is vulnerable to symbolic execution with constraint solvers. To thwart symbolic executions, Sharif *et al.* notice that some code blocks can be concealed by setting a trigger condition with a one-way function, so that the constraint solver cannot solve [40]. Wang *et al.* propose another obfuscation technique to combat the symbolic execution by exploring the general limitation of symbolic execution tools in analyzing loops. Their idea is to employ unsolved conjectures [45] to confuse the termination condition of loops. The approach is vulnerable when the tricks of unsolved conjectures are recognized. Other than setting tricks on the source code, Linn *et al.* propose to obfuscate the binaries directly by inserting some error bits, which can be automatically corrected during execution by the CPU but not by the current disassembly tools [27]. The security of such a protection is very limited and vulnerable to dynamic analysis, *i.e.,* the actual instruction trace can be easily obtained once the software is being executed. To deter dynamic analysis with debuggers, Oishi *et al.* propose to engage some camouflaged anti-debuggers, which, however, is not effective for homemade debuggers. On protecting software from tampering, another general popular approach is to detect the unauthorized modifications during runtime by employing a self-checksumming mechanism [8, 21]. The self-checksumming mechanism applies redundantly overlapped checksum testers inside the program to verify its integrity. On the other side, several investigations focus on defeating the protections [31, 36, 46, 49]. Wurster *et al.* propose to defeat the self-checksumming approach with a duplicated memory attack, and examine its effectiveness on several popular CPU types [46]. Qiu *et al.* propose to identify the self-checksumming code using taint analysis approaches [36]. Yadegari *et al.* find a more general way of deobfuscating an obfuscated algorithm [49]. Our work is different from all the existing work in that we focus on impeding the replication of software tampering, which is not yet properly addressed.

5.3 Software Diversification

The idea of software diversity is initially proposed for fault tolerance or software reliability engineering, which is known as N-version programming [9, 28]. Cohen in [12] firstly proposes to create functionally equivalent programs to enhance software security. Forrest *et al.* in [19] also state that the beneficial effects of diversity in computing systems have been overlooked, and introducing diversities into computer systems can make them more robust to replicated attacks. They propose several possible ways to create such diversities with respect to the program behavior, including adding nonfunctional code, refactoring code, or diversifying the memory layout. Crane *et al.* in [15] build upon fine-grained code diversification to prevent code-reuse attacks. They adopt function permutation [23], register allocation randomization, and callee-saved register save slot reordering [34] in the diversification process. However, to our best knowledge, all these investigations do not consider to automatically generate functionally nonequivalent programs to improve tamper-resistance as a holistic approach.

6. CONCLUSION

This work focuses on impeding the replication of software tampering, which is a unique perspective for anti-tampering research. We propose the N-version obfuscation (NVO) approach, which automatically generate and deliver functional nonequivalent software versions to different machines. In this way, the original tampering approach, which could be either software repacking or dynamic injection, will not adapt to work on machines other than the attacker's experimental one. To demonstrate its applicability in practical scenarios, we propose a candidate solution for general networked applications. Specifically, the candidate solution introduces functional nonequivalence by adding or using a MAC mechanism. Our evaluation result shows that the achieved functionally nonequivalent diversities can be effective against tampering replication, and the complexity to tamper the software system is linearly increased with the number of software versions, which can be automatically generated with trivial cost.

Although the NVO idea is promising, this work can be extended in various ways. Our candidate solution highly depends on the characteristic of network communications and the MAC mechanism. More solutions are expected in the future to help us explore the technique more thoroughly. Besides, our candidate solution simply incorporates existing anti-tampering approaches. A systematic study on how to effectively combine them are needed. Finally, the candidate solution has not been examined publicly, and its security should be further improved with real applications.

Acknowledgements

This work was supported by the National Basic Research Program of China (973 Project No. 2014CB347701), the Key Project of National Natural Science Foundation of China (Project No. 61332010), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14205214 of the General Research Fund), and 2015 Microsoft Research Asia Collaborative Research Program (Project No. FY16-RES-THEME-005). Yangfan Zhou is the corresponding author.

7. REFERENCES

[1] Android Auto. *https://www.android.com/auto/.*

[2] DexGuard. *https://www.guardsquare.com/dexguard.*

[3] FIPS Pub 180-4: Secure Hash Standard (SHS). *http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.*

[4] IDA. *https://www.hex-rays.com/products/ida/.*

[5] ProGuard. *http://developer.android.com/tools/help/proguard.html.*

[6] A. Appel. Deobfuscation is in np. *Princeton University, Aug*, 21:2, 2002.

[7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. In *CRYPTO*, pages 1–18. Springer, 2001.

[8] H. Chang and M. J. Atallah. Protecting software code by guards. In *Security and Privacy in Digital Rights Management*, pages 160–175. Springer, 2002.

[9] L. Chen and A. Avizienis. N-version programming: a fault-tolerance approach to reliability of software operation. In *Proc. the 8th IEEE International Symposium on Fault-Tolerant Computing*, pages 3–9, 1978.

[10] Y. Chen, R. Venkatesan, et al. Oblivious hashing: A stealthy software integrity verification primitive. In *Information Hiding*, pages 400–414. Springer, 2003.

[11] S. Chow, P. Eisen, H. Johnson, and P. C. V. Oorschot. White-box cryptography and an aes implementation. In *Selected Areas in Cryptography*, pages 250–270. Springer, 2003.

[12] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.

[13] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, 1997.

[14] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Trans. on Software Engineering*, 28(8):735–746, 2002.

[15] S. Crane, C. Liebchen, et al. Readactor: Practical code randomization resilient to memory disclosure. In *Proc. of the 36th IEEE Symposium on Security and Privacy*, volume 15, 2015.

[16] D. Dunaev and L. Lengyel. Complexity of a special deobfuscation problem. In *Proc. of the 19th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 1–4, 2012.

[17] E. Eilam. *Reversing: secrets of reverse engineering.* John Wiley & Sons, 2011.

[18] P. Faruki, V. Laxmi, V. Ganmoor, M. S. Gaur, and A. Bharmal. Droidolytics: robust feature signature for repackaged android apps on official and third party android markets. In *Proc. of the 2nd IEEE International Conference on Advanced Computing, Networking and Security*, pages 247–252, 2013.

[19] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proc. of the 6th IEEE Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[20] M. N. Gagnon, S. Taylor, and A. K. Ghosh. Software protection through anti-debugging. 2007.

[21] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Security and Privacy in Digital Rights Management*, pages 141–159. Springer, 2002.

[22] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-llvm-software protection for the masses. 2015.

[23] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proc. of the 22nd IEEE Annual Computer Security Applications Conference*, pages 339–348, 2006.

[24] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Proc. of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[25] P. Larsen, S. Brunthaler, and M. Franz. Automatic software diversity. 2015.

[26] L. Lei, Y. Wang, J. Zhou, D. Zha, and Z. Zhang. A threat to mobile cyber-physical systems: Sensor-based privacy theft attacks on android smartphones. In *Proc. of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 126–133, 2013.

[27] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of the 10th ACM Conference on Computer and Communications Security*, pages 290–299, 2003.

[28] M. R. Lyu et al. *Handbook of software reliability engineering.* IEEE Computer Society Press, 1996.

[29] M. R. Lyu and Y.-T. He. Improving the n-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, 42(2):179–189, 1993.

[30] J. Ming, D. Xu, L. Wang, and D. Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 757–768, 2015.

[31] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. of the 23rd IEEE Annual Computer Security Applications Conference*, pages 421–430, 2007.

[32] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, 86(1):176–186, 2003.

[33] P. O'Kane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. 2011.

[34] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proc. of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[35] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. Identifying and understanding self-checksumming defenses in software. 2015.

[36] J. Qiu, B. Yadegari, B. Johannesmeyer, et al. A

framework for understanding dynamic anti-analysis defenses. In *Proc. of the 4th ACM Program Protection and Reverse Engineering Workshop*, 2014.

[37] C. Ren, K. Chen, and P. Liu. Droidmarking: Resilient software watermarking for impeding android application repackaging. In *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 635–646, 2014.

[38] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, pages 44–60. Springer, 1998.

[39] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proc. of the 30th ACM Annual Computer Security Applications Conference*, pages 56–65, 2014.

[40] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proc. of the 15th Annual Network & Distributed System Security Conference (NDSS)*, 2008.

[41] T. Shields. Anti-debugging: a developers view, 2010.

[42] J. Shu, J. Li, Y. Zhang, and D. Gu. Android app protection via interpretation obfuscation. In *Proc. of the 12th IEEE International Conference on Dependable, Autonomic and Secure Computing*, 2014.

[43] M. Sun, M. Li, and J. Lui. Droideagle: seamless detection of visually similar android apps. In *Proc. of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 9, 2015.

[44] H. Wang, Y. Guo, Z. Ma, and X. Chen. Wukong: a scalable and accurate two-phase approach to android app clone detection. In *Proc. of the ACM International Symposium on Software Testing and Analysis*, pages 71–82, 2015.

[45] Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In *Proc. of the 16th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2011.

[46] G. Wurster, P. V. Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proc. of the 26th IEEE Symposium on Security and Privacy*, 2005.

[47] H. Xu, Y. Zhou, C. Gao, Y. Kang, and M. R. Lyu. Spyaware: Investigating the privacy leakage signatures in app execution traces. In *Proc. of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

[48] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, volume 15, pages 732–744, 2015.

[49] B. Yadegari, B. Johannesmeyer, et al. A generic approach to automatic deobfuscation of executable code. Technical report, 2015.

[50] F. Zhang, D. Wu, P. Liu, and S. Zhu. Program logic based software plagiarism detection. In *Proc. of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2014.

[51] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *Proc. of the 4th ACM Conference on Data and Application Security and Privacy*, pages 199–210, 2014.

[52] W. Zhou, X. Zhang, and X. Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proc. of the 8th ACM Symposium on Information, Computer and Communications Security*, pages 1–12, 2013.

[53] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. of the 2nd ACM Conference on Data and Application Security and Privacy*, pages 317–326, 2012.

[54] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of the 33th IEEE Symposium on Security and Privacy*, 2012.