StIns4CS: A State Inspection Tool for C#

Amjad Ibrahim
Technische Universität München
Boltzmannstr. 3
85748 Garching bei München, Germany
ibrahim@cs.tum.edu

Sebastian Banescu
Technische Universität München
Boltzmannstr. 3
85748 Garching bei München, Germany
banescu@cs.tum.edu

ABSTRACT

Software protection aims to prevent unauthorized use, analysis, modification and distribution of software. This goal is hard to achieve, especially for a program running on a platform (e.g. physical device) controlled by an adversary also known as man-at-the-end (MATE) attacker. Self-checking is one technique for protecting the integrity of software by having the code check itself.

In this paper, we present the design and implementation of a self-checking tool called StIns4CS. Our tool implements self-checking via state inspection by source code transformations of programs written in the C# language. More specifically, StIns4CS augments code by adding runtime checkers to it. We discuss the effectiveness of StIns4CS by implementing attacks targeting our approach, and measuring different aspects of the effectiveness, stealth and cost of the protection. Based on the evaluation we show the trade-off between the efficiency and effectiveness of StIns4CS in protecting software against unauthorized modification. We propose an approach to improve stealth of the code added by StIns4CS and we show further improvements of stealth by combining self-checking with virtualization obfuscation.

CCS Concepts

•**Security and privacy** → **Software security engineering;**

Keywords

Software protection, Tamper proofing, MATE attacks

1. INTRODUCTION

According to Falcarin et al. [6], software protection can be divided into four categories: (1) *obfuscation* which hampers reverse engineering, (2) *tamper-proofing* which hampers unauthorized modification of code, (3) *watermarking* which allows tracking programs and (4) *birthmarking* which enables plagiarism detection. This paper is chiefly concerned with the second category, i.e. *tamper-proofing* which aims to ensure that a program maintains the input-output behavior intended by its developer during runtime. This goal is motivated by its commercial impact. Statistics from the *Business Software Alliance* and the *International Data Corporation* show that the retail value of pirated software globally is $63.4 billion in 2013 [8].

The goal of maintaining behavior integrity of an application is hard to achieve when this application must run on untrusted platforms, i.e. platforms controlled by malicious users. Malicious users are also known as *man-at-the-end* (MATE) attackers and they have many more attack vectors than the *man-in-the-middle* (MITM) attacker proposed by Dolev and Yao [5]. For instance, MATE attackers can analyze the memory contents of an application during execution via interactive debuggers. Moreover, they can even tamper with the memory contents and even the code of the application. This can be achieved using freely available tools like OllyDbg [1], or reverse engineering tools like ILSpy [2].

The approach developed in this paper, focuses on protecting the integrity of pure functions (without side-effects) via a technique called *state inspection* described in [4]. Such functions are found in security-sensitive components, but also components that process important assets (e.g. virtual currency). For example, Figure 1 shows an implementation of a function implementing a form of role based access control, which checks if the current user has a certain permission passed as an input argument. The permission check can be disabled by a MATE attacker, who modifies the code such that line 9 always returns true. Another domain where checking pure functions can be effective is the financial domain. For instance, Figure 2 shows an example where a MATE attacker tampers with the upper part of the code that calculates the sum of a set of financial transactions. The attacker can add a rounding function, as seen in Figure 2 bottom part (line 5), that "leaks" small amounts of money to his account. Our approach will detect such a tampering attack. However, our approach will not detect tampering attacks which do not affect the return value of the function being attacked, e.g. disabling or adding a print statement. Also applying our approach to functions with side-effects may cause unwanted application behavior.

The implementation presented in this paper, identifies unauthorized modifications to programs, such as the one from Figure 1, and reacts via a certain *response mechanism* (e.g. stopping program execution). State inspection is performed by:

SPRO'16 October 28 2016, Vienna, Austria

ISBN 978-1-4503-4576-7/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2995306.2995313

[1] http://www.ollydbg.de/
[2] http://ilspy.net/

```
1 public bool HasPermission(string reqPermission){
2  bool bFound = false;
3  foreach (UserRole role in this.Roles){
4    bFound = (role.Permissions.Where(
5    p => p.PermissionDes == reqPermission) > 0);
6    if (bFound)
7      break;
8  }
9  return bFound; // replace by return true
10 }
```

Figure 1: Permission check example.[4]

```
1 public float sum(float [] arr){
2  float sum = 0;
3  for (int i = 0; i < arr.Length; i++){
4        sum += arr[i];
5  }
6 return sum;
7 }
```

```
1 public float sum(float [] arr){
2  float sum = 0;
3  for (int i = 0; i < arr.Length; i++){
4        // subtracts 0.00005
5        arr[i] = round(arr[i]);
6        sum += arr[i];
7 }
8 return sum;
9 }
```

Figure 2: Calculation tampering example.

(1) generating a set of input-output pairs denoted T, for a set of deterministic functions in the program, then (2) making calls to these functions during runtime using inputs from T and (3) checking if the outputs of the function match the outputs in T corresponding to the given input. If the output of a function f does not match the expected output from T, then we assume that f has been modified by a MATE attacker.

This paper makes the following contributions:

- Presents the design and implementation of StIns4CS, a tool that adds tamper-detection via state inspection to programs written in the C# language [3].
- Proposes using symbolic execution to generate the set of input-output pairs for a certain set of functions.
- Presents an evaluation of the performance and stealth of StIns4CS with various response mechanisms.
- Proposes a way to increase the stealth of the response mechanism by degrading results.

The remainder of this paper is organized as follows. Section 2 describes the design and implementation of StIns4CS. Section 3 presents the evaluation of StIns4CS, w.r.t. both performance and effectiveness against attacks. Section 4 presents related work. Finally, we will conclude the paper in Section 5 and discuss possible directions of future work.

[3]https://github.com/tum-i22/stins4cs

[4]http://www.codeproject.com/Articles/875547/Custom-Roles-Based-Access-Control-RBAC-in-ASP-NET

```
1 // Create an instance of the target class
2 A instance = new A();
3 // Invoke the checked function "F1"
4 var output = instance.F1();
5 // Compare against known value
6 if (output != expectedValue){
7       callResponse();
8 }
```

Figure 3: Guard example.

2. DESIGN AND IMPLEMENTATION

This section presents the design and implementation of StIns4CS, including a new response mechanism to raise the bar against pattern matching attacks.

2.1 Code Guard Networks

StIns4CS protects source code by adding *guards*, each consisting of a few lines of code, illustrated in Figure 3. Each guard is added to one function which is called a *checking function*. Guards are invoked during run-time execution of the function in which they reside. They are responsible for detecting if the code of another function (called *checked function*) has been tampered with. The guard code in Figure 3 starts by creating an instance of the class (`A`) in which the checked function resides (line 2). Then, it calls the checked function `F1` (line 4). Guards compare the return value of the checked function against a precomputed expected value (line 6). If the values are different then a response mechanism is invoked (line 7).

Let each function that contains a guard, be a node in a directed graph. Arcs in this graph are added starting from each checking function to its corresponding checked function. Since the same function could be both checker and checked, there may be cycles in the graph. We call this graph a *guard network*. The idea of creating a network of code guards was presented by Chang & Atallah [2]. Their concept, called *code introspection*, is based on checking the static representation of the program, i.e the code. The downside of code introspection is its stealthiness, since it is not common to have program read their own code. Therefore, it is relatively easy to identify guards that use code introspection even if the program is heavily obfuscated [17], compared to the approach of state inspection, which we use in this paper.

An example of a guard network is illustrated in Figure 4, where the functions are A.F1, which checks B.F2, which checks C.F3, which checks D.F4, which checks A.F1. The advantage of a guard network which consists of strongly connected components (i.e. each node is reachable from every other node), is that all functions are being checked by all other functions in that component. Therefore, if an attacker tampers with any of the functions in a strongly connected component, this modification is detected when one of the functions in that component is executed.

The number of nodes in a strongly connected component (called *component size*) is configured by the user of StIns4CS. For instance the *component size* in Figure 4 is equal to 4. In section 3 we will see how this configuration affects both the level of protection and performance of applications.

2.2 Stack Inspection

One problem with the idea of creating networks of guards

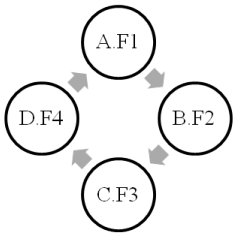

Figure 4: Guard network consisting of 4 functions.

```
1 String stack = Environment.StackTrace;
2 if (!stack.contains("A.F1")){
3   A instance = new A();
4   Out = instance.F1();
5   if (out != expectedValue){
6         callResponse()
7   }
8 }
```

Figure 5: Stack inspection solution

that contain cycles as the example from Figure 4, is that it may lead to an infinite sequence of checks, that will eventually cause an application crash due to a stack-overflow error. This problem could be solved in multiple ways, e.g. a global variable could be added as a flag indicating when a function call is performed by a checker. In this case other checks can be stopped by verifying this flag. However, an attacker who is aware of the implementation of StIns4CS, could identify this global flag and always set it such that no checks would ever be performed. Another problem with this solution is thread safety. Any parallel threads will not be tested since the flag will be set as busy.

We propose an alternative solution by inspecting the call stack before executing a guard. If the same function appears twice on the call stack, then we avoid performing the check, because a cycle was detected. Figure 5 shows a snippet of the checker with this solution. A stack inspection statement is added to each check (line 2), this statement ensures that the checked function is not already on the stack. The rest of the check line 3-8 is the regular check code. This solution helps in breaking guards cycles, but the downside is that for some cases (e.g. pairs of mutually invoking functions), checks will not be carried out.

Since all code guards perform stack inspection, it may seen as a single point of failure. However, our implementation replicates the (compact) code for stack inspection for all individual guards. As shown in Figure 5, each guard first calls `Environment.StackTrace` or `System.Diagnostics.StackTrace` or any other methods, to obtain a string representation of the stack. Afterwards it searches if the function to be checked is within this string. One way of attacking the stack inspection is to hook all the library calls which can be used to obtain a string representation of the stack and make sure that the function always returns a string containing the name of the function to be checked. Our current implementation cannot defend against function hooking. However, multiple types of hooking can be detected in numerous ways [13], which is complementary to this work. Moreover, the MATE attacker cannot hook any function calls in stack inspection statements, because s/he would break the functionality of other parts of the application which also use those functions. This is what would happen if the MATE attacker hooked the (`contains()`) function on line 2 of Figure 5 such that it always returns `true`. Such an attack will affect the behavior of all the code statements that are using `contains()` on string variables, which we assume is commonly used in any program. Another way of attacking stack inspection is to add a statement that appends the name of the function in the if-statement on line 2 of Figure 5, to the string assigned in line 1. However, this is not a genuine "single point of failure", because the MATE would have to find all of the guards and add such a statement to each of them.

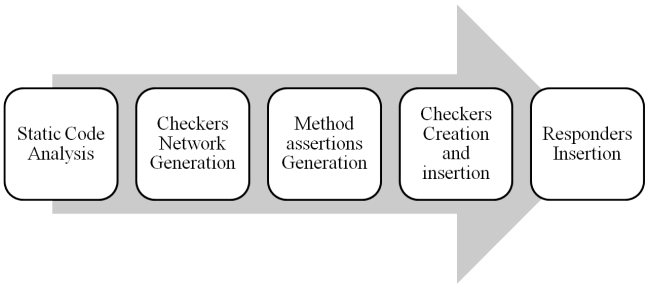

Figure 6: Steps of transformation

2.3 Code Transformation Workflow

In this section we describe the workflow of how StIns4CS transforms C# source code. The 5 steps of the workflow are shown in Figure 6. The first step analyzes the static code (subsubsection 2.3.1) to gather information about the code, e.g. the functions names. Next, a network of check relations is generated (subsubsection 2.3.2). In the third step the assertions are generated (subsubsection 2.3.3). Next, the checkers are finalized and added to the code (subsubsection 2.3.4). Finally, the response mechanism is inserted into the code (subsubsection 2.3.5). We use the *Roslyn* open-source C# compiler framework, to process the source code in all of the 5 steps of the workflow [5,6].

2.3.1 *Static code analysis*

The aim of this step is to determine all the namespaces, types, functions, properties (members that provide a flexible mechanism to read write data in C#), fields in the source code given as input to StIns4CS. All classes in the source code are traversed to collect information about each function. A data structure of all the public functions is maintained. Information about return types, parameters, method visibility (public, private), comments and annotations is stored for use in the following steps.

2.3.2 *Checker network generation*

After analyzing the code, random subsets of functions are chosen. These subsets will form strongly connected components in the guard network, as displayed in Figure 4.

In this step we must account for functions that depend on the state of the application during run-time and side effects of functions. We do this through annotations [16] added by the developer of the C# program and/or the user of StIns4CS,

[5]https://github.com/dotnet/roslyn
[6]https://roslyn.codeplex.com/

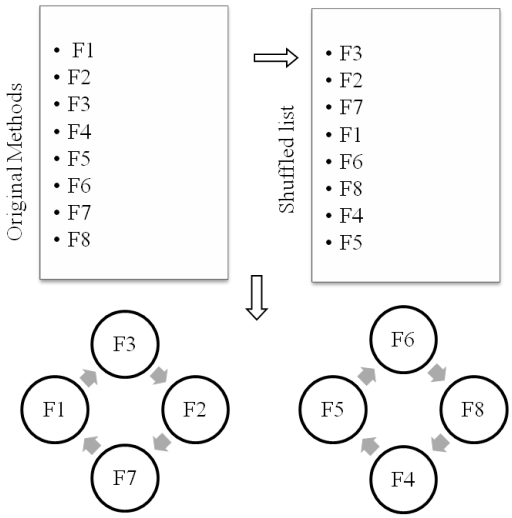

Figure 7: Steps of creating checking networks

which exclude such methods from being used in the runtime assertions.

Figure 7 shows the steps of creating checking networks. In the top-left part of the figure we have the list of original functions from the source code. Firstly, we create a randomly shuffled copy of the list of original functions (upper-right of Figure 7). Secondly, we partition into lists of size less or equal to *component size* (n), this number is chosen by the user. Finally, each function from the resulting lists checks the following function in the list and the last function checks the first function in the list. This leads to two four-nodes networks in our example from Figure 7 (bottom).

2.3.3 Method assertion generation

This step generates the challenge and expected result using an automated unit tests generator, based on symbolic execution, developed by Microsoft, called Pex [9, 12, 20, 21]. Utilizing Pex for this step is effective, because Pex tries to cover all the branches of the code. Pex usually generates multiple test cases for the same method, we use the last one, which is generally non-trivial. However, we could easily extend the current implementation, to use all of the generated test cases across multiple checks.

Although Pex tries to provide help in creating instances of the classes that are checked, there are cases where Pex cannot achieve this goal due to complexity of the code. For example, if the target class has an *interface* as a constructor parameter, then Pex will not be able to pick an implementation of that interface to use for the instantiation. To overcome such cases, StIns4CS will create a factory class that can help in creating those instances.

2.3.4 Checker creation and insertion

Using the list of methods and code information from the first step of the workflow in Figure 6, the checker network from step two, and the test cases (input-output pairs) from step three, the checking code is created in this phase. The part of the code that is generated, as seen in Figure 3, is as follows:

- Code to create an instance of the target class (line 2).
- Invoking the target method and comparing the result to the expected value (line 4-6).
- Calling the response mechanism (line 7).

After generating this code snippet, it is inserted at the begging of the checking method body.

2.3.5 Responder insertion

Intuitively, response mechanisms punish misbehaving users. Punishment can be performed by, for instance, crashing the system or halting the execution of the application. In general, manual and application-specific techniques have been used to add responders and the focus of the research was to detect the tampering. Tan et al. [19] propose a tamper-response system that raises the bar of detecting the checkers, by introducing delayed, probabilistic failures in a program. The main technique is to corrupt certain parts of the program's internal state at well chosen locations. Forcing the program either to fail or exhibit degraded performance (deliberate injection of programming bugs like "array out of bounds" errors).

Sometime, the punishment mechanism itself leads attackers to discover the protection. However, we argue that response mechanisms can be used to monitor tampering acts. To achieve this, response mechanisms can be logging agents that gather information about the tampering like, cracked features, time and location. Such information can help software producers in their efforts to harden their protection schemes. Also, this advances the efforts of creating accountable systems where misbehaving users can be detected.

The response in StIns4CS is configurable, i.e. the user selects the action that should be triggered when tamper is detected. The mechanisms that are supported by StIns4CS are:

- Immediate crash, which terminates the process and gives the underlying operating system the specified exit code.
- Random delayed crash, which sets a timer to exit the application. The interval is set to a random number of seconds. However, the user can configure the upper bound of the interval.
- Remote logging, using log4net[7]. Log statements are sent to a remote server. The format of the log statements can be set by the user.
- Do nothing.

Response mechanisms can be mixed in any way using a weighting factor, for example 20% do nothing, 80% immediate crash. This ability to mix response mechanisms raises the bar against attackers who want to discover the checking statements by backtracking analysis, starting from the point where the response was observed.

2.4 Primitive combination

We introduce another kind of response mechanism called *primitive combination*, along the 4 mechanisms described previously. However, the current implementation only works for functions that return primitive data types. *Primitive combination* operates by sabotaging the returned values of functions, rather than invoking a direct action. This is done by

[7] https://logging.apache.org/log4net/

```
1 public bool isEmpty(string s) {
2   bool testbool = s.Any();
3   return testbool;
4 }
```

```
1 public int getLength(string s) {
2   int length = s.Length;
3   return length;
4 }
```

Figure 8: Functions to be used for primitive combination.

```
1 public bool isEmpty(string s) {
2   A instance = new A();
3   int i = instance.getLength("test");
4   bool testbool = s.Any();
5   return testbool && (i == 4);
6 }
```

Figure 9: Primitive combination of functions in Figure 8.

"incorporating" the result of the checked method with the return value of the checking method. *Primitive combination* removes: (1) the comparison of the result of the method call with the expected value (line 6 in Figure 3) and (2) the direct call to the response mechanism (line 7 in Figure 3), which improves the stealth of the check. This association also produces different forms of combinations based on the different data types. So, *primitive combination* improves the protection level by adding diversity to the protection code. Which thwarts generalizing an automatic pattern matching attack that depends on patterns targeting the direct calls to the response.

The concept is to *combine* results of two functions: the checking and the checked functions, given that they both return primitive type data. The combination is implemented in a way that affects the checker's result, only if the result of the checked function is not as expected. Thus, after combination, the checker function will return wrong values or perhaps throw exceptions in the case of tampering with the checked function. The primitive types of C# are: byte, sbyte, int, uint, short, ushort, long, ulong, float, double, decimal, char, bool, string [8]. The first column of Table 1 lists the possible combinations between C# primitive types and the proposed combination of them. The second column of Table 1 indicates the return value of the checker function, the third column indicates the return value of the checked function and the fourth column indicates how the return value of the checker (denoted $rvChecker$) is combined with the actual return value of the checked function $rvChecked_{act}$, depending on the expected return value of the checked function $rvChecked_{exp}$.

Example: The checking function returns boolean and the checked function returns integer as seen in Figure 8. The result of combining these functions is seen in Figure 9. In Figure 9, lines 2-3 are similar to the normal check. Line 5 shows the actual combination, where we create a new return statement that combines the results based on Table 1.

[8]https://msdn.microsoft.com/en-us/library/ms228360(v=vs.90).aspx

2.5 Limitations of StIns4CS

The current implementation of StIns4CS has some limitations, which we describe in this section. Most of the limitations are technical, and can be eliminated as part of a future work.

Firstly, StIns4CS does not transform classes with function overloading. Methods with the same name in the same class will not be transformed as expected. The main reason behind this limitation, is the fact that Pex generates a unique name for the method, and this makes it hard to directly map to the source code as the case with other functions. The workaround is to annotate methods with the same name to be ignored. The solution is to rename methods having the same name in the same class by doing a preprocessing of the input source code.

Secondly, inner classes are not supported by StIns4CS, any inner class will not be transformed. Inner classes are copied to the resulting code, but they will not be tamper-proofed. Currently, the first public class in a source file will only be transformed. This limitation can be eliminated by adapting the tool to account for inner classes. For simplicity, we built the tool under the assumption that each source file will contain one class.

Lastly, the self-checking approach that StIns4CS follows is based on verifying the return value of the function. The current implementation only verifies pure functions. StIns4CS does not verify *output parameters* of functions. These parameters should also be verified similarly to the return value.

3. EVALUATION

For the purpose of evaluating the effectiveness of StIns4CS, we analyzed three aspects of our technique. Firstly, the *effectiveness* against attacks, i.e. we measure how well the protection holds against manual static and dynamic code-tampering attacks. Secondly, the *stealth* of guards inside the code, which is a measure of how well the checkers blend with the original code against pattern matching attacks. Lastly, the *cost* of checking, which measures the overhead added to the application by the checkers.

Experiments, were performed on three test sets of C# code:

1. Primitive Apps: we developed this test set to showcase the *primitive combination* feature of StIns4CS. It contains different functions that return different types of data values, like integer, double or string. The complete code base contains 12 methods written in 161 lines of code.

2. Pex samples: a collection of simple code snippets offered by the Pex project documentation [9]. This code base contains some known implementation of different algorithms such as: *BinarySearch, QuickSort* and other 10 algorithms. Each algorithm is implemented in its own class. The advantage of these samples is that they contain different useful functions from different areas. A simple graphical user interface was added to the samples, to make it easier to run the samples and invoke them. The complete code base contains 14 methods written in 603 lines of code.

3. SGML Reader [10] : a C# .NET library for parsing HTML/SGML files using the XmlReader API. This is a real-

[9]http://research.microsoft.com/en-us/projects/pex/documentation.aspx
[10]https://github.com/MindTouch/SGMLReader

Table 1: The possible combinations between C# primitive types

Combination type	Checker function return type	Checked function return type	Combination of return values
Numeric-Numeric	Numeric	Numeric	$rvChecker + (rvChecked_{exp} - rvChecked_{act})$
Numeric-Bool	Numeric	Bool	$rvChecker + (int)(rvChecked_{exp} \oplus rvChecked_{act})$
Numeric-Char	Numeric	Char	$rvChecker + (rvChecked_{exp} - rvChecked_{act})$
Numeric-String	Numeric	String	$rvChecker + (int)(rvChecked_{exp} \neq rvChecked_{act})$
Bool-Numeric	Bool	Numeric	$rvChecker \oplus (rvChecked_{exp} \neq rvChecked_{act})$
Bool-Bool	Bool	Bool	$rvChecker \oplus (rvChecked_{exp} \neq rvChecked_{act})$
Bool-Char	Bool	Char	$rvChecker \oplus (rvChecked_{exp} \neq rvChecked_{act})$
Bool-String	Bool	String	$rvChecker \oplus (rvChecked_{exp} \neq rvChecked_{act})$
Char-Numeric	Char	Numeric	$(char)(rvChecker + (rvChecked_{exp} - rvChecked_{act}))$
Char-Bool	Char	Bool	$(char)(rvChecker + (int)(rvChecked_{exp} \oplus rvChecked_{act}))$
Char-Char	Char	Char	$(char)(rvChecker + (rvChecked_{exp} - rvChecked_{act}))$
Char-String	Char	String	$(char)(rvChecker + (int)(rvChecked_{exp} \neq rvChecked_{act}))$
String-Numeric	String	Numeric	$rvChecker.Substring((rvChecked_{exp} - rvChecked_{act}))$
String-Bool	String	Bool	$rvChecker.Substring((int)(rvChecked_{exp} \oplus rvChecked_{act}))$
String-Char	String	Char	$rvChecker.Substring((rvChecked_{exp} - rvChecked_{act}))$
String-String	String	String	$rvChecker.Substring((int)(rvChecked_{exp} \neq rvChecked_{act})$

world application contains 22 public methods and a total of 4953 lines of code.

3.1 Effectiveness against code tampering

Here we aim to verify how the three test sets of C# code applications protected using StIns4CS respond to either static code tampering or dynamic memory tampering. Moreover, we show how guards can be disabled manually, and the effect of disabling guards.

3.1.1 Input parameters of StIns4CS

For this experiment, we varied the input parameter *component size* (denoted *n*) of StIns4CS by giving it values: 2, 4, 8 and 16. The *component size* determines the number of checks that will be triggered by each execution of a checker function. This affects when tampering is detected, i.e. a larger value of *component size* results in a larger number of functions that could detect code tampering of another function. For example, if the *component size* is 2, then we have pairs of functions that check each other.

The response mechanism was set to immediate crash because it is deterministic (unlike randomly delayed crash), and can be easily noticed during testing (unlike primitive combination). Although we acknowledge that some responses are harder to detect by an attacker (e.g random delayed crash, primitive combination), we consider the worst case for the defender, in this experiment.

3.1.2 Static tampering attack

The steps to implement the *static tampering attack* are as follows:

1. Decompile the executable of the C# application using ILSpy [11].
2. Edit the decompiled code by using Reflexil a .NET assembly editor [12]. A tampering attack was performed by modifying an arithmetic operators used by a checked function, e.g. replaced an integer addition by a multiplication.
3. Run the C# application again with the patched code and check how and when the program reacts to the tampering.

3.1.3 Dynamic tampering attack

The steps to implement the *dynamic tampering attack* are as follows:

1. Attach a debugger to the running application. We have used the Visual Studio 2015 debugger [13].
2. Find a point in the code loaded in process memory which could be the target of a tampering attack.
3. Pause the debugger and manipulate the memory where the targeted code is located. For instance, we located the x86 assembly instruction `inc` and replace it with a (`nop`) instruction.
4. Continue debugging and check behavior of the patched program.

3.1.4 Attack results

By repeating the previous static and dynamic tampering attacks several times, we found that modifying checked functions is discovered by *at least n-1* other functions. In other words, the guards will detect tampering when any of the functions in a checking network is invoked. In addition, we may have cases where different connected components are linked by function calls in the original (un-protected) code. This leads to having even more that *n-1* functions which can detect code tampering attacks.

3.1.5 Disabling guards

An interesting case for both the static and dynamic tampering attacks is to target the code of guards such that they are disabled, i.e. can no longer detect tampering of the checked

[11] http://ilspy.net/
[12] http://reflexil.net/
[13] https://msdn.microsoft.com/en-us/library/sc65sadd.aspx

Table 2: The regular expressions used for checking stealth of code guards.

ID	Regular expression syntax
Regex1	`if\s*\(((?!\s*\{).+)\)\s*\{?(.|\s)*?\}?`
Regex2	`if\s*\(!Environment.StackTrace((?!\s*\{).+?)\)\s*\{?(.|\s)*?\}?`
Regex3	`(\s*Environment.Exit)`

function. We *disabled* one guard in a strongly connected component and verified if other guards are still working. We found out that disabling one guard is possible for a MATE attacker performing a static or dynamic attack as described in subsubsection 3.1.2, respectively subsubsection 3.1.3.

MATE attacks which disable a guard by modifying its code (see Figure 3) to disable the comparison on line 6 or the call to the response mechanism on line 7, are not detected by our implementation. If one guard is disabled, the rest of the checking function in the same checking network still detect tampering of other code. Depending on the topology of the network and the location of the disabled guard, the effectiveness is reduced, because some cycles are broken, which leads to unconnected components. However, this attack is only possible if the attacker locates the checks, which is why stealth of the checking mechanism is important. Note that if the *primitive combination* response mechanism is employed, then lines 6 and 7 from Figure 3 are not needed, as we saw in the example from Figure 9. In subsection 3.2 we will see different ways of how we can improve the stealth of guards.

3.2 Stealth of the checking code

For the purpose of evaluating the stealth of the guards, we implemented a pattern matching attack against code protected with StIns4CS. For the purpose of the stealth evaluation we use the Pex samples described in the beginning of section 3. This attack consists of the following steps:

1. Decompile the executable of the C# application using ILSpy.
2. Locate guards by traversing the decompiled files and parses the content to verify the existence of the predefined regular expressions (regex) from Table 2.
3. Output the number of occurrences, the line where the occurrence started and the size (in characters) of the match.

Figure 10 shows an example, where the script found 4 occurrences in the C# source code file called *BinarySearch.cs*, of the regular expression `Regex1` from Table 2. For example, line 4 specifies the line number of the first occurrence (`Line: 12`) and the length of the match (`Length: 15`).

3.2.1 Test parameters

We verified the existence of three different patterns, shown in Table 2, in the protected code. The first pattern targets any general if-statement (`Regex1`). The second regular expression (`Regex2`) targets any if-statement which contains a call to `Environment.StackTrace`, i.e. the stack inspection operation we use in guards to prevent stack overflow errors (see subsection 2.2). The third regular expression (`Regex3`) targets a specific response mechanism, namely immediate

```
1 Result For: pexsamples-regex1 matched with regex
2 if\s*\(((?!\s*\{).+)\)\s*\{?(.|\s)*?\}?
3 4 matches found in: BinarySearch.cs
4 Line: 12 Length: 15
5 Line: 20 Length: 16
6 Line: 33 Length: 18
7 Line: 40 Length: 19
```

Figure 10: A sample of the report generated by the matching script.

crash. However, `Regex3` could be easily extended to also target random delayed crash and remote logging. By testing these three patterns we cover the parts that can be targeted with a pattern matching attacks. Other parts of the checker are dynamic parts and they blend in with the code, like class instantiation, or method invocation.

We tested different versions of the protected code, by varying two aspects that affect the code of the guards. The first aspect is, whether or not to use the *primitive combination* as a response mechanism. The second is, whether or not to obfuscate the code using a *virtualization obfuscation* tool for C# source code called VOT4CS[14]. We used this second aspect to measure the extra protection that obfuscation can bring to our implementation.

3.2.2 Test procedure

For each test case in Table 3, the following procedure was followed:

1. Generate protected and possibly obfuscated code using the settings indicated in the first 3 columns from Table 3.
2. Build the source code, generate the binary, for example a class library (DLL).
3. Decompile the binary code into source code using ILSpy.
4. Run the pattern matching script against the decompiled code for `Regex1`, `Regex2`, and `Regex3`.

3.2.3 Tests results

For each version of the protected code we calculated *accuracy of matching* as a metric to show the level of matching a regular expression can achieve over the protected code. It is a ratio between the number of correctly matched checks (true positives and true negatives) to the total number of checks added to a project:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN},$$

where **TP** is the number of true positives, which represents the correctly matched checks. **TN** is the number of true negatives, which represents the number of not-matched non-checks portions of code. **FP** is the number of false positives, which represents the incorrectly matched checks. **FN** is the number of false negatives, which represents the incorrectly missed checks.

The results we obtained are plotted in Figure 11. In *Test case 1*, the attack identified two parts of the guard, that are

[14] https://github.com/tum-i22/vot4cs

Table 3: Test cases settings and their accuracy percentage

Name	`Primitive_Combination`	`Virtualization`	`Regex1 Any if-statement`	`Regex2 Stack inspection if`	`Regex3 Response call`
Test case 1	No	No	15	100	100
Test case 2	YES	No	15	100	50
Test case 3	No	YES	0	0	100
Test case 4	YES	YES	0	0	50

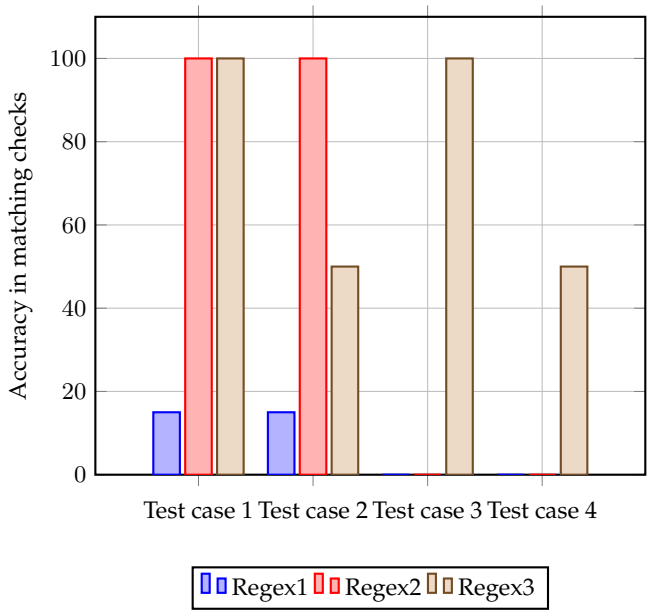

Figure 11: Comparison of accuracy of matching attacks.

vulnerable to pattern matching attacks. The first part is the stack inspection statement, while the second part is the response statement. For the response statement, we have already presented ideas that improve the stealth of this part. Namely, using primitive combination, which removes the part of comparing expected and actual return values along with the response statement. This can be seen in the results of `Regex3` in *Test case 2*, where 50% of the guards were transformed using *primitive combination*. On the other hand, the vulnerability of stack inspection motivates the idea of obfuscating the guards. Therefore, we have carried out an experiment to evaluate the protection that virtualization obfuscation can bring. The results in *Test cases 3* and *Test case 4* show that pattern matching of `Regex1` and `Regex2` drop to 0. The 50% of guards that can still be identified by `Regex3` in *Test case 4* could not be transformed using *primitive combination* because their return values are not of primitive types. This motives the need for future work in extending the idea of *primitive combination* to non-primitive data types, i.e. objects.

3.3 Cost of checking

Since augmenting the code with guards adds function invocations to the program during run-time, the performance of the C# application is affected. The exact performance degradation depends on the performance of the input code itself. The *component size* input parameter of StIns4CS, also affects the overhead, because it controls the number of added calls for each function invocation.

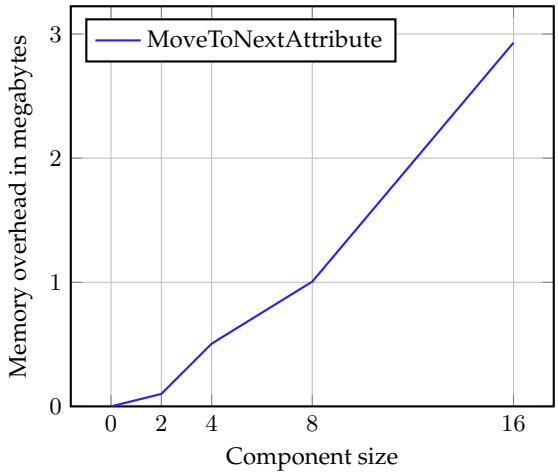

Figure 12: Memory allocation impact for SGML Reader

3.3.1 *Memory and execution time overhead*

For measuring the impact on memory allocation and execution time, we randomly selected one function called *MoveToNextAttribute* from the SGML Reader application and executed it 10 times. Each value recorded is the plots given in this section are obtained by averaging 10 readings. Plot Figure 12, shows the memory allocation values, for the *component size* input parameter varying from 0 to 16 (x-axis of Figure 12). As seen in Figure 12, memory allocation is directly proportional to the number of nodes in the strongly connected components. The *component size* represents the number of checks that are invoked when a function is executed, and hence more calls will lead to more memory consumption.

Similarly, plot Figure 13 shows the execution time for the same function with different values for the *component size* input parameter on the x-axis. As opposed to memory allocation, execution time is logarithmic with respect to the *component size*. The number of nodes in the checking network represents the number of checks that are invoked when a function is executed, and hence more calls will lead to a higher execution overhead.

As expected, execution time and memory allocation increase with the component size, for a given input. Unless StIns4CS is configured to ignore functions that can be considered "hot code", there is no guarantee on how much more memory or execution time, a guard will consume.

3.3.2 *Code size*

The transformation workflow adds lines of code to the original code base. Unlike previous performance measures, the size of code is not affected by the *component size*, because each function in the selected list of methods to be protected will

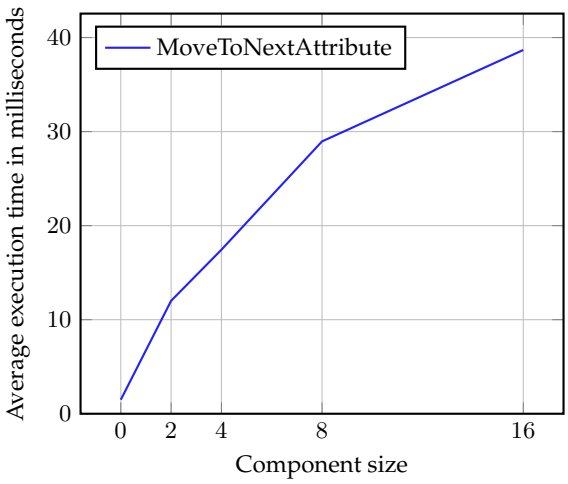

Figure 13: Elapsed time impact for SGML Reader

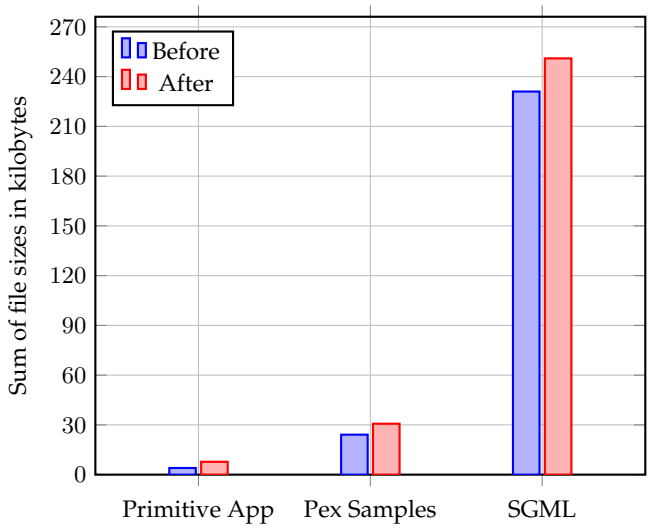

Figure 14: Impact on file size

contain exactly one code guard, regardless of the *component size*. The size of the resulting code depends on the number of functions that are selected for protection by StIns4CS. Figure 14 shows the increase in file size for each of the test sets of C# code.

4. RELATED WORK

Since in this paper we implemented both a defense mechanism for self-checking and an attack on it, the related work is divided accordingly, into defenses and attacks.

4.1 Defenses

Chang and Atallah [2] propose building a network of code regions, where a region can be a block of user code, a checker, or a responder. In this method checkers check each other in addition to user code by comparing a known checksum of piece of code to runtime checksum of the same code. If the checker has discovered that a region has been tampered with, a responder will replace the tampered region with a copy stored elsewhere. An important aspect of this algorithm is that it is not enough for checkers to check just the code, they must check each other as well. If checkers are not checked, they are easy to remove. Horne et al. [10] build on top of [2], by hiding the expected (precomputed checksum) value which is easy to identify, because of its randomness. The idea is to construct the checksum function such that unless the code has been tampered with, the function always checksums to a known number (usually zero). Having this function allows to insert an empty slot within the region under protection, and later give this slot a value that makes the region checksum to zero. The technique of Horne et al. [10] randomly places large numbers of checkers all over the program, but makes sure that every region of code is covered by multiple checkers. To minimize pattern-matching attacks, this method describes how to generate a large number of variants of lightweight checksum functions. The disadvantage of the *code introspection* approach used by both [2] and [10] is its stealthiness, because code that reads itself is seldom used for other purposes. StIns4CS uses state-inspection which is not subject to this disadvantage.

Chen et al. [3] propose an idea called *oblivious hashing*, where the checksum value is computed over the *execution trace* rather than the static code. The checksum can be computed by inserting instructions that monitor changes to variables and the execution of instructions. A problem with automating this technique, is that it is hard to predict what side effects a function might have. It might destroy valuable global data or allocate extraneous dynamic memory that will never be properly freed. Furthermore, there is a problem with nondeterministic functions that depend on the time of day, network traffic, thread scheduling, and so on, because they do not have a fixed output that can be checked. This technique also faces the issue of automatically generating challenge data (test inputs) that most of the code of a function. StIns4CS implements a variant of oblivious hashing therefore it also suffers from the same disadvantages. However, we address the last issue by proposing the use of symbolic execution in order to generate the challenge data.

Jacob et al. [11] propose an approach which depends on a unique property of the x86 instruction set architecture (ISA). The x86 ISA has a *variable instruction length* (1-15 bytes) with no alignment, this means instructions can start at any offset in the code. This results in the possibility of having overlapping or even nested instructions. So the basic idea will be that when a block is executed it computes a checksum of another block. For the purpose of protecting the code, we need two blocks to share instruction bytes. Having two blocks to share instruction bytes, can be achieved by interleaving the instructions and inserting jumps to maintain semantics. The advantage in this technique is that the code checksumming computations will not require reading the code explicitly. The disadvantage is mainly the performance overhead of the added instructions. Jacob et al. [11] report that the protected binary can be up to three times slower than the original. Even though this overhead may be acceptable in many circumstances, this technique cannot be applied to programs that execute on the Common Language Runtime such as programs written in C#.

Cappaert et al. [1] propose a technique that hinders both code analysis and tampering attacks simultaneously through code encryption. During run-time, code decryption can be done at a chosen granularity (e.g. one function at a time),

when that part of code is needed at run-time. This technique performs integrity-checking of the code by using it to compute the keys for decryption and encryption. The basic idea is using the checksum value of a function, as the decryption key of another function. The advantage of this technique is that the encryption key is computed at run time, which means the key is not hard-coded in the binary and therefore hard to find through static analysis. The disadvantage of this technique is the run-time overhead as well as the its stealth.

Martignoni et al. [14] and Seshandri et al. [18] propose establishing a *trusted computing base* to achieve verifiable code execution on a remote un-trusted system. The trusted computing base in the two methods is established using a verification function. The verification function is composed of three components: (i) a checksum function, (ii) a send function, and (iii) a checksum function. However, the main difference between the two methods is the checksum function. In the work of Martignoni et al. [14] generates a new checksum function each time and sends it encrypted to the untrusted system. In the work of Seshandri et al. [18], the checksum function is known a priori and the challenge issued by the dispatcher consists in a seed that initializes this function. Since the remote component in both methods knows precisely in which execution environment the function must be executed and knows the hardware characteristics of the un-trusted system, it can compute the expected checksum value and can estimate the amount of time that will be required by the un-trusted system to decrypt and execute the function, and to send back the result. Since Intel x86 architecture, the architecture for which the approach of Seshandri et al. [18], was developed, is full of subtle details, researchers have found ways to circumvent the remote component. Also, a limitation of the approach of Martignoni et al. [14], is the impossibility to bootstrap a tamper-proof environment on simultaneous multi threading (SMT) or simultaneous multi processing (SMP) systems. On such systems, the attacker can use the secondary computational resources (parallel threads for example) to forge checksums or to regain control of the execution after attestation.

4.2 Attacks

Pattern matching attacks as we have described in Section 3 are not the only possible attacks against self-checking defenses. Wurster et. al. [22] defeated all approaches based on code introspection by using a modified operating system. The patched operating system replicated memory pages containing program code, so that data reads and instruction fetches at the same virtual address access different physical addresses. The attack created a virtual Harvard memory architecture with distinct instruction and data memories. Self-modification of the code was used in [7,15] to protect self-checking against the memory split attack.

Qiu et al. [17] proposed another attack on code introspection approaches using dynamic taint analysis. The attack relies on the fact that introspection based approaches read their own code, process it, compare it to some value and then perform a conditional jump based on the result. Therefore, they propose tainting the code segment in memory and tracing the program during execution. Afterwards, they detect which branches depend on tainted data, because those branches are the checkers which must be bypassed by an attacker. Since our approach is not based on code introspection it is not vulnerable to any of the attacks presented here.

5. CONCLUSIONS

This paper presents the design and implementation of a self-checking tool called StIns4CS, which protects applications against unauthorized modification using state inspection. We propose using symbolic execution to generate the set of input-output pairs needed for state inspection. We also present a way of improving the stealth of state inspection by a technique we call *primitive combination*. We performed an case-study based evaluation using several software samples, to validate the effectiveness and efficiency of StIns4CS.

For the purpose of the effectiveness evaluation, we used static and dynamic analysis attacks, that aim to either tamper with the protected application, or discover the locations of the checking mechanism. The effectiveness evaluation indicated that code protected by StIns4CS is effective against tampering attacks on the original code. Moreover, increasing the size of the strongly connected components of functions which check each other, increased the number of functions that detect tampering and hence increase the protection level.

Stealth of the checking code is a critical aspect in the success of any self-checking technique. During the effectiveness evaluation we noted that stealth is important for raising the bar against pattern matching attacks, which aim to disable checks. The concept of *primitive combination* we proposed in this direction, is a promising start. However, there is need for future work in order to extend this idea to objects. Moreover, we also employed virtualization obfuscation and showed that the stealth of the checks improves further.

Finally, our study also included a performance evaluation of the protected applications, where we show a relation between how StIns4CS is configured and the performance of the protected application. Although the impact of self-checking using StIns4CS depends mainly on the nature of the code and the performance of its own functions, the value of the *component size* has an important impact on both memory and execution time.

Future work

One of the major findings during the stealth analysis of the tamper-proofed code was the influence of stack inspection. This part of each checker is added as a way to prevent stack overflow due to the cyclic relationship between checkers. This needs to be further studied to inspect other ways of breaking cycles of execution in the checking network. One possible idea is to extend the code analysis step to include information from the code call graph. The added information can be used to find networks with the longest paths without creating a cycle.

Primitive combination is an interesting to enhance the stealth of the check. However, its application is conditioned by the existence of two functions that return primitive type values. An interesting path would be to study generalizing this concept to apply it to other non-primitive types.

6. ACKNOWLEDGMENTS

The authors would like to thank Alexander Pretschner and Benjamin Kräemer for their valuable feedback and insights.

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

7. REFERENCES

[1] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. De Bosschere. Towards tamper resistant code encryption: Practice and experience. *lecture notes in Computer Science*, 4991:86–100, 2008.

[2] H. Chang and M. J. Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2002.

[3] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *Information Hiding*, pages 400–414. Springer, 2003.

[4] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[5] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[6] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski. Guest editors' introduction: Software protection. *Software, IEEE*, 28(2):24–27, 2011.

[7] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Computer Security Applications Conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[8] Globalstudy.bsa.org. Bsa global software survey: The compliance gap: Home, 2013.

[9] P. Godefroid, P. De Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *Software, IEEE*, 25(5):30–37, 2008.

[10] 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.

[11] M. Jacob, M. H. Jakubowski, and R. Venkatesan. Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *Proceedings of the 9th workshop on Multimedia & security*, pages 129–140. ACM, 2007.

[12] K. Jamrozik, G. Fraser, N. Tillman, and J. De Halleux. Generating test suites with augmented dynamic symbolic execution. In *Tests and Proofs*, pages 152–167. Springer, 2013.

[13] J. Leitch. Iat hooking revisited, 2011.

[14] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40. Springer, 2010.

[15] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[16] M. D. Network. Attributes (c# programming guide).

[17] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. Identifying and understanding self-checksumming defenses in software. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 207–218. ACM, 2015.

[18] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *ACM SIGOPS Operating Systems Review*, 39(5):1–16, 2005.

[19] G. Tan, Y. Chen, and M. H. Jakubowski. Delayed and controlled failures in tamper-resistant systems. In *In Proceedings of 8th Information Hiding Workshop*. Citeseer, 2006.

[20] N. Tillmann and J. De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. Springer, 2008.

[21] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *Software, IEEE*, 23(4):38–47, 2006.

[22] G. Wurster, P. C. Van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Security and Privacy, 2005 IEEE Symposium on*, pages 127–138. IEEE, 2005.