Poster: A Measurement Framework to Quantify Software Protections

Paolo Tonella, Mariano Ceccato
Fondazione Bruno Kessler
Trento, Italy
{tonella,ceccato}@fbk.eu

Bjorn De Sutter, Bart Coppens
Ghent University
Belgium
{Bjorn.DeSutter,Bart.Coppens}@elis.ugent.be

ABSTRACT

Programs often run under strict usage conditions (e.g., license restrictions) that could be broken in case of code tampering. Possible attacks include malicious reverse engineering, tampering using static, dynamic and hybrid techniques, on standard devices as well as in labs with additional special purpose hardware equipment. ASPIRE (http://www.aspire-fp7.eu) is a European FP7 research project devoted to the elaboration of novel techniques to mitigate and prevent attacks to code integrity, to code/data confidentiality and to code lifting. This paper presents the ongoing activity to define a set of metrics aimed at quantifying the effect on code of the ASPIRE protections. The metrics have been conceived based on a measurement framework, which prescribes the identification of the relevant code features to consider and of their relationships with attacks and protections.

1. MEASUREMENT FRAMEWORK

This paper presents the ASPIRE plan and ongoing work about using code metrics to estimate the impact of protections on code. To this aim, we have defined a measurement framework that extends the Goal-Question-Metric approach [1] and we have instantiated it in the ASPIRE context. The goal and the questions that we consider depend on the protections that the project is elaborating and on the attacks that are relevant for such protections. Starting from attacks and protections, we identified what are the relevant features that metrics should capture quantitatively. Our measurement framework consists of the following steps:

1. [**goal**] *Succinct definition of the goal the metrics are to achieve in the quantification of protections*
2. [**questions**] *Expansion of the goal into a set of questions that will be answered once metrics values are available*
3. [**measurable features**] *Identification of a set of measurable features, based on protections and attacks, relevant for answering the questions*
   - 3.1. [**protections**] *Enumeration of the protections whose strength is to be quantified by metrics*
   - 3.2. [**attacks**] *Enumeration of the attacks whose difficulty of realisation is to be quantified by metrics*
   - 3.3. [**coverage**] *Iterative definition of measurable features based on the coverage matrices: protections × features and attacks × features*
4. [**metrics**] *Derivation of metrics from the measurable features*

Step 3.3 is particularly critical, since it aims at producing the list of measurable features from which metrics are derived. This step is conducted iteratively. Initially, based on protections and attacks, a first list of measurable features is produced. Then, coverage matrices are computed to associate protections/attacks and features. In the protection × feature matrix, an entry equal to 1 indicates that there is a direct, measurable effect of the protection on the considered feature. The entry is 0 otherwise. In the attack × feature matrix, an entry equal to 1 indicates that the attack is obstructed by the (massive) presence of the feature. Whenever these two matrices indicate poor coverage of protections/attacks, new measurable features must be defined. Whenever a feature does not contribute to coverage of any protection/attack, the feature is removed from the list.

In the next sections we instantiate the framework for the ASPIRE protections.

2. GOAL

Software protection aims at increasing the cost incurred to mount a successful attack, so as to make it not economically convenient, typically by augmenting the time required to complete an attack. At the same time, software protection aims at minimal execution overhead associated with its protections. We can thus state the general goal of the metrics as follows:

> ***Goal*** *of the metrics is to quantify the benefits and costs associated with the adoption of the software protections. On one hand, we want to obtain indications about the increased effort incurred by an attacker when trying to tamper with the protected program. On the other hand, we want to measure the execution overhead introduced by the ASPIRE protections.*

Metrics should provide approximate indications about the effort increase that can be expected, hence they are in the category of the *complexity metrics*. Even if such metrics

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
ACM 978-1-4503-2957-6/14/11.
http://dx.doi.org/10.1145/2660267.2662360.

cannot be easily turned into attack time, they provide useful indications about the amount of code elements that are made more complex to analyse for an attacker. The metrics values give a clear picture of the amount of protection introduced into the code, relatively to the original sensitive code base and with respect to alternative configurations of the same protection. Users of the ASPIRE protections will obtain precise, quantitative indications about the impact of each protection in the code.

The costs for the protections are quantified by *performance metrics*, since the protection logics will necessarily introduce some execution overhead. We measure also such negative impact of the protections, so as to provide the ASPIRE users with the full quantitative picture, including measurable benefits (increased code complexity) and measurable costs (runtime overhead).

3. QUESTIONS

From the general goals that guide the definition of the measurement framework, we derived a set of specific questions that metrics are expected to be able to answer:

- **Q1**: How large and how complex is the portion of program that contains protections to be defeated by the attacker?
- **Q2**: How much does the protected program differ from the unprotected (hence, easily attackable) program?
- **Q3**: What is the runtime overhead incurred due to the ASPIRE protections?

Protections might be applied to a subset of the whole program. Question **Q1** deals with the size of the protected part. In fact, the larger is the portion subject to protection, the more difficult is for the attacker to locate the specific place where to start an attack. In case the protected part is very small, the remaining *unprotected* part could be still sensitive and exposed to easy attacks.

Question **Q2** refines question **Q1** by taking into account the distance between protected and unprotected code. The larger the complexity gap between the original unprotected code and the protected code, the more difficult it is for the attacker to understand and modify the program under attack. In fact, assuming that unprotected code can be easily attacked, the search space for the attacker consists mostly of the differential code that is introduced by the protections.

While questions **Q1** and **Q2** deal with the benefits of the code protections, question **Q3** deals with its costs, which are basically performance degradation costs.

4. MEASURABLE FEATURES

In the following, we describe the scope in which metrics are defined. Such a scope consists of the protections under development in the ASPIRE project and of the attacks that the ASPIRE protections intend to block or delay. For each protection and attack in the scope of ASPIRE we determined the associated measurable features. A *measurable feature for a protection* represents the impact on the source/binary code associated with the application of the protection to a program. A *measurable feature for an attack* consists of the source/binary code elements that represent an obstacle for the attackers and that must be circumvented to mount a successful attack.

The ASPIRE protections fall into the following groups:

PT$_1$: *Data hiding.* Sensitive program data are obfuscated in the code.
PT$_2$: *Renewability.* Code portions are periodically replaced at run time, so as to prevent the reuse of attack knowledge gathered on previous versions of the program.
PT$_3$: *Splitting.* Code portions are isolated from the main program and executed on a trusted device or host, which communicates with the main client to query and update program state.
PT$_4$: *VM obfuscation.* Code portions are executed in obfuscated form inside a special purpose obfuscated virtual machine.
PT$_5$: *Remote attestation.* Sensitive code portions are periodically checked for integrity, in response to a remote attestation request coming from a trusted device or host.
PT$_6$: *Binary obfuscation.* Code obfuscation at the binary level.

The attacks in the scope of ASPIRE can be classified as follows:
AT$_1$: *Static structural code and data recovery.* Reverse engineering of abstract, structured program representations from the source/binary code. In the scope of this attack are only static approaches to reverse engineering.
AT$_2$: *Structural matching of binaries.* Attackers match the program code against known or related code fragments to identify procedures relevant for the attack. A variant consists of computing the difference between program versions to identify the security patches applied to the older version.
AT$_3$: *Static tampering attacks.* The attacker modifies the code to disable protections and security checks.
AT$_4$: *Attacks on communication channels.* The attacker can observe and manipulate the messages exchanged over the network as well as the client code generating such messages.
AT$_5$: *Fuzzing.* The program is executed by a fuzzing tool for automated input data generation. The execution is monitored for security errors and information leakage.
AT$_6$: *Debugging.* The program's execution is interrupted at break points or stepped and program variables are inspected and modified through a debugger.
AT$_7$: *Dynamic structure and data analysis.* The attacker recovers the program's control flow and data flow by means of dynamic information obtained from execution traces.
AT$_8$: *Dynamic tampering.* The attacker modifies the execution of the program by means of code injection, debuggers, emulators or binary code rewriting.
AT$_9$: *Hybrid analysis.* The attacker combines static and dynamic analyses to retrieve relevant information about the program.

Attacks considered here include both manual human attacks and automated attacks using tools. Thus, measurable features should capture not only the *potency* of protection against manual human attack, but also the *resilience* against automated attacks.

We have iteratively refined a list of relevant measurable features based on the ASPIRE protections and attacks. Specifically, we eventually obtained the following measurable features that are affected by the ASPIRE protections and that obstruct the attacks to the ASPIRE protections:
MF$_1$: *Code size.* Larger programs are potentially more difficult to attack.
MF$_2$: *Chains between variable definitions and variable usages.* Data origin is more difficult to identify if the data

dependencies are more complicated in the program.
MF$_3$: *Complexity of the control flow.* Reverse engineering of the program structure is more difficult if the control flow of the program is made more complicated.
MF$_4$: *Access to data structures and functions through chains of pointers.* Data manipulation and control flow management are difficult to reverse engineer if they involve extensive pointer computations.
MF$_5$: *Invocation of functions.* The call graph structure can potentially disclose relevant information to attackers, so a more complex call graph structure may represent a higher barrier for attackers.
MF$_6$: *Size of the execution traces.* Dynamic analysis is more difficult if long execution traces have to be analysed.
MF$_7$: *Variable-value propagation at run time.* Complex data definition chains at run time make it more difficult for an attacker to detect the origin of a value in the program.
MF$_8$: *Control flow complexity upon execution.* The dynamic structure of the program provides an abstract program representation which helps attackers; hence, making it more complex can potentially increase the attack effort.
MF$_9$: *Complexity of dynamically allocated data structures.* Use of complex, pointer intensive data structures makes the dynamic recovery of information about such data structures more complicated.
MF$_{10}$: *Dynamic call stack.* A complex dynamic structure of the invocation graph may reduce the possibility of program comprehension during debugging.

Other relevant measurable features deal with the performance impact of a protection, which may represent an obstacle to its adoption. We identified the following performance features:
PF$_1$: *Execution time.* The execution time of protected code is expected to be higher than the execution time of the original code, because of the additional protection logics.
PF$_2$: *Memory allocated to the program at run time.* Additional data structures are often required to implement the protection logics, which increases the memory footprint of the protected program in comparison with the original program.

The selected measurable features provide a reasonably good coverage of protections and attacks, so we deem them as an adequate starting point for the definition of metrics.

5. PRELIMINARY METRICS

From the measurable features, we have derived a list of preliminary metrics. Metrics fall into two categories: (M1) static metrics; (M2) dynamic metrics.

M1: The static ASPIRE metrics measure the static difference between the protected program P' and the unprotected program P. The measurement is taken either on the source or the binary code, and depends on various code elements in which P and P' may differ:
DST$_s$ *Delta STatements*: number of statements in which the source/binary code of P and P' differ.
DDD$_s$ *Delta Data Dependencies*: number of def-use pairs in which the source/binary code of P and P' differ.
DCD$_s$ *Delta Control Dependencies*: number of conditional/loop control dependencies in which the source/binary code of P and P' differ.
DPT$_s$ *Delta PoinTers*: number of points-to relations (between pointers and pointed locations) in which the source/binary code of P and P' differ.
DCG$_s$ *Delta Call Graph*: number of call relations in which the source/binary code of P and P' differ.

M2: The dynamic ASPIRE metrics measure the dynamic difference between the protected program P' and the unprotected program P. The measurement is taken on a set of execution traces reporting the sequence of source or binary code statements executed in the considered execution scenarios:
DST$_d$ *Delta STatements*: number of statements in which the source/binary execution traces for P and P' differ.
DDD$_d$ *Delta Data Dependencies*: number of def-use pairs in which the source/binary execution traces for P and P' differ.
DCD$_d$ *Delta Control Dependencies*: number of conditional/loop control dependencies in which the source/binary execution traces for P and P' differ.
DPT$_d$ *Delta PoinTers*: number of points-to relations (between pointers and pointed locations) in which the source/binary execution traces for P and P' differ.
DCG$_d$ *Delta Call Graph*: number of call relations in which the source/binary execution traces for P and P' differ.
DET *Delta Execution Time*: difference between the execution time of P' and P.
DMS *Delta Memory Size*: difference between the memory allocated for the execution of P' and P.

To obtain the dynamic traces necessary to compute the dynamic metrics M2, a set of test cases or executable scenarios are needed. Such test cases may be either provided with the program to be protected, or it can be generated automatically, e.g. by random or search based test case generators, or by dynamic symbolic executors. Test cases should exercise the protected portions of the program (test cases that do not exercise those program portions can be excluded from consideration, since they do not contribute to the difference). When using automated test case generators, it is possible to configure them so that they explicitly target the statements in the protected program portions.

6. CONCLUSION AND FUTURE WORK

We have presented a measurement framework that can be used for the definition of metrics in the context of software protection. Metrics are expected to quantify the impact of protections on the code complexity, as well as the increased difficulty of an attack. To obtain metrics with such properties, we iteratively refined them based on two coverage matrices: protections $\times$ features and attacks $\times$ features.

In our future work, we will complete the derivation of the metrics from the measurable features, we will develop a tool to compute them and we will validate them empirically on the systems considered as case studies in the ASPIRE project.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 609734.

7. REFERENCES

[1] V. R. Basili. Software modeling and measurement: the goal/question/metric paradigm. 1992.