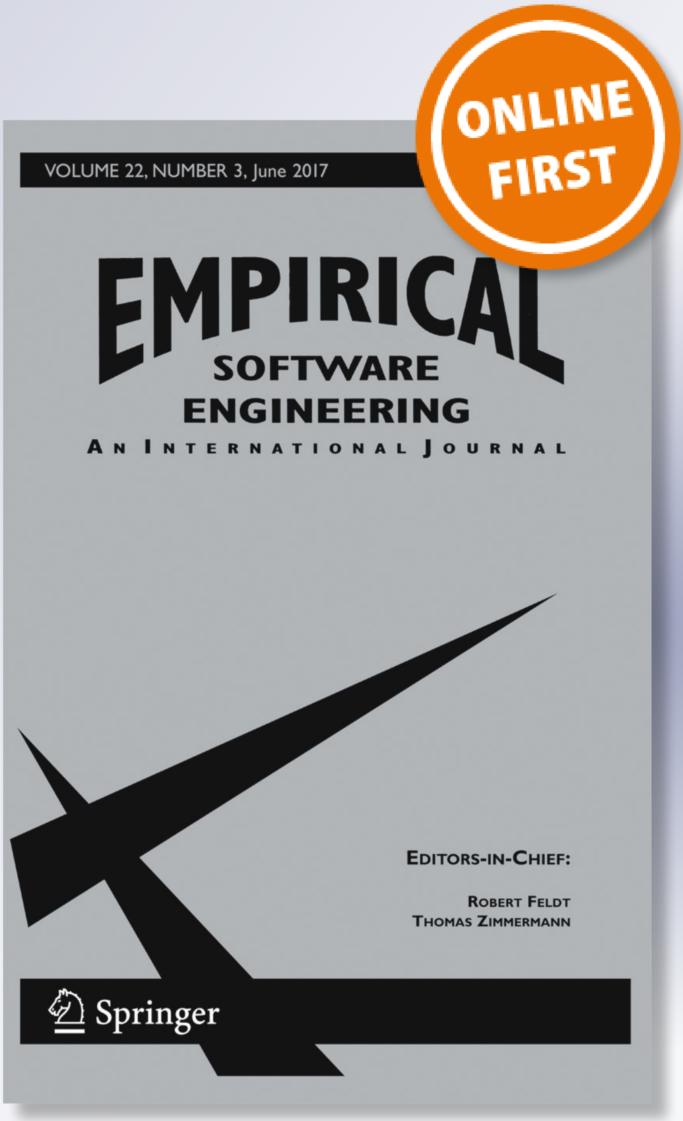
Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge

Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens & Bjorn De Sutter

Empirical Software Engineering
An International Journal

ISSN 1382-3256

Empir Software Eng
DOI 10.1007/s10664-018-9625-6



Springer

Empir Software Eng
https://doi.org/10.1007/s10664-018-9625-6

Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge

Mariano Ceccato[1] · Paolo Tonella[1] · Cataldo Basile[2] ·
Paolo Falcarin[3] · Marco Torchiano[2] · Bart Coppens[4] ·
Bjorn De Sutter[4]

Abstract When critical assets or functionalities are included in a piece of software accessible to the end users, code protections are used to hinder or delay the extraction or manipulation of such critical assets. The process and strategy followed by hackers to understand and tamper

Communicated by: David Lo and Alexander Sebrenik

Note on order of authors: The last five authors participated to open coding, conceptualization and paper writing. So did the two first authors, which also designed the qualitative analysis and led the execution thereof. They also led the professional hacking experiment design and process. The last two authors also led the design and execution of the Public Challenge experiment.

✉ Mariano Ceccato
ceccato@fbk.eu

Paolo Tonella
tonella@fbk.eu

Cataldo Basile
cataldo.basile@polito.it

Paolo Falcarin
falcarin@uel.ac.uk

Marco Torchiano
marco.torchiano@polito.it

Bart Coppens
bart.coppens@ugent.be

Bjorn De Sutter
bjorn.desutter@ugent.be

[1] Fondazione Bruno Kessler, Trento, Italy

[2] Politecnico di Torino, Turin, Italy

[3] University of East London, London, UK

[4] Universiteit Gent, Gent, Belgium

Published online: 26 May 2018

with protected software might differ from program understanding for benign purposes. Knowledge of the actual hacker behaviours while performing real attack tasks can inform better ways to protect the software and can provide more realistic assumptions to the developers, evaluators, and users of software protections. Within Aspire, a software protection research project funded by the EU under framework programme FP7, we have conducted three industrial case studies with the involvement of professional penetration testers and a public challenge consisting of eight attack tasks with open participation. We have applied a systematic qualitative analysis methodology to the hackers' reports relative to the industrial case studies and the public challenge. The qualitative analysis resulted in 459 and 265 annotations added respectively to the industrial and to the public challenge reports. Based on these annotations we built a taxonomy consisting of 169 concepts. They address the hacker activities related to (i) understanding code; (ii) defining the attack strategy; (iii) selecting and customizing the tools; and (iv) defeating the protections. While there are many commonalities between professional hackers and practitioners, we could spot many fundamental differences. For instance, while industrial professional hackers aim at elaborating automated and reproducible deterministic attacks, practitioners prefer to minimize the effort and try many different manual tasks. This analysis allowed us to distill a number of new research directions and potential improvements for protection techniques. In particular, considering the critical role of analysis tools, protection techniques should explicitly attack them, by exploiting analysis problems and complexity aspects that available automated techniques are bad at addressing.

Keywords Software protection · Software hacking · Attack process · Attack model · Attack taxonomy · Empirical study · Hacker model · Reverse engineering · Analysis tools · Code understanding · Defeat protections

1 Introduction

Software running on the client side is increasingly security-critical. Mobile apps are more and more used for sensitive functionalities (e.g., bank account management) and a similar trend can be observed for web apps, due to recent frameworks (e.g., Angular) that move most of the processing to a single page residing in the browser. Among others, the sensitive operations that may run on the client side include authentication, license management, and Intellectual Property Rights (IPR) enforcement.

As a consequence, client apps (and the underlying libraries) contain critical assets that must be secured. The state of the art approach to secure them is software protection. However, software protection is intrinsically a difficult task and the techniques available in practice cannot defeat an attacker with an arbitrarily high amount of time and resources available (e.g., theoretical results (Barak et al. 2001) show that perfect obfuscation is impossible in the general case). Hence, the assessment of software protections must take into account an economical trade off between the attack difficulties that protections introduce and the expected remuneration for a successful attack. Effective protections are protections that make such a trade-off not convenient for the attacker.

The EC funded Aspire project (https://aspire-fp7.eu) has developed a software protection tool chain that spans many different categories of techniques, including white box cryptography, diversified cryptographic libraries, data obfuscation, non-standard virtual machines, client-server code splitting, anti-callback stack checks, code guards, binary code obfuscation, code mobility, anti-debugging, and remote attestation.[1] The authors of this paper are the academic project partners and their task was to investigate, develop, and validate novel protection techniques.

The empirical investigation consisted of two experiments. In the first experiment, the three industrial partners of Aspire applied the protection tool chain to their respective industrial case studies and evaluated its effectiveness with the help of professional penetration testers. The outcome of this experiment consisted of three reports, including detailed narrative information about the tasks conducted in the attempt to defeat the Aspire protections. The reports describe the activities carried out during the attacks, the encountered obstacles, the followed strategies, and the difficulty of performing each task. They represent a unique opportunity to investigate the behaviour of real-world expert hackers while carrying out an attack.

The second experiment was a Public Challenge. It consisted of eight small applications that were protected by the Aspire tool chain and were published on-line, offering one prize per application to the first attacker able to successfully complete a given attack task. Five applications were successfully broken, as it happened by the same amateur hacker. That winner was interviewed over email to collect, in several iterations, detailed information about his attack activities, encountered obstacles, followed strategies, and attack difficulty. The responses in the emails were bundled into a single report concerning general information about the hacker, his general impressions about the challenges, and accounts on the five broken applications, which was then further analysed.

The reports collected in the two experiments include invaluable knowledge on the way hackers understand protected software. We applied a systematic qualitative analysis procedure to the reports. In particular, data from the first experiment with industrial hackers were analysed using open coding, conceptualization, and model inference. Instead, data from the second experiment (i.e. the Public Challenge) were analysed with closed coding, conceptualization, and model inference, to extend the taxonomy of concepts related to hackers activities, available after the first experiment.

The overall results of these experiments are a taxonomy of concepts and models of the hacker behaviour that can be used by protection developers as a qualitative support to validate the assumptions under which breakage of a protected asset is deemed economically disadvantageous. In fact, relevant factors that determine the attack strategies might be overlooked if an idealized hacker model that departs substantially from the reality is assumed. Our model of the hacker behaviour, which is grounded on observations obtained from real attack tasks and supported by evidence collected in the field, provides a solid basis on the expected attacker behaviour to protection developers. Moreover, our work identified a number of research directions and guidelines that could substantially improve the practice of software protection. We have consolidated such lessons learned in the form of a research agenda for software protection.

Despite the execution of multiple experiments in diversified contexts, we acknowledge that we did not achieve theoretical saturation. However, this does not undermine the validity of the presented approach, of our implementation, and of the current results. It is anyway the case that theoretical saturation of attacker tasks has a limited temporal validity, because this subject, being an arms race, is a moving target. New stronger attacks are elaborated at a constant pace, and this motivates to conceive brand new protections, that are later subject

[1] The Glossary presents brief descriptions of these techniques and of concepts and terms introduced later in the paper.

to even more powerful attacks. Additional experiments are needed to achieve a wider comprehension of this phenomenon and reach saturation regarding the current state of the art, and will be needed in the future to keep up with this evolving topic. The proposed methodology supports the inclusion of data from additional experiments, as this paper already demonstrates with the inclusion of the second experiment.

It is worth noting that a study of this size (three and five attacked software, three industrial teams and one practitioner) has quite limited general validity, as acknowledged in Section 4 with our threat to validity analysis. Nonetheless, data collected in these experiments are interesting and worth sharing with other researchers. Moreover, the preliminary results obtained in our qualitative study are expected to trigger future quantitative studies, with stronger external validity, which may confirm or refute our claims. One contribution of this paper is to set the key research questions, the key concepts and the reference models for such future studies.

This paper extends a conference paper that presented the first experiment with industrial case studies and its results (Ceccato et al. 2017). This paper adds the second experiment, i.e., the public challenge. This extension is important for several reasons. First, it allows us to broaden the coverage of the resulting taxonomy and models to more attack scenarios: to a wider range of attackers, with different levels of expertise and different preferences in terms of attack approaches, to a wider range of assets and protected applications, and to more combinations of deployed protections. These extensions are important because defenders of software assets need to defend against all possible attack paths by all possible attackers at once. Secondly, it allows us studying the differences in behavior observed in the two experiments, and the extent to which the resulting taxonomy and models are impacted by including additional experiments. In addition to the extension towards more attack scenarios, we also extended the original taxonomy with the software elements that the hackers in both experiments considered in their activities. This extension is useful to let defenders focus on the concrete features their protections need to impact.

The paper is organized as follows. The qualitative analysis methodology used to annotate the hacker reports is described in Section 2. The extracted taxonomy and models are presented in Section 3. We discuss the research directions for the improvement of existing protections and for the development of new ones that emerged from our studies in Section 4. Section 5 is devoted to related work, while Section 6 concludes the paper and sketches our plans for future work.

2 Qualitative Analysis Method

We have collected qualitative data from two sources: (1) Professional Hackers; (2) Public Challenge Winner. Professional hackers were involved in the attempt to break the protections of three industrial applications, provided by the three industrial partners of Aspire. The Public Challenge consists of eight small programs protected by some combinations of Aspire protections. Those programs were made available on the challenge website to registered users. For each program, the first hacker able to break it was granted a prize.

We used two distinct annotation methods with the two experiments, because an initial taxonomy of concepts, produced from the first experiment with professional hackers, was available at the time when the public challenge experiment was conducted. In particular, we have applied different coding approaches to the professional hacker reports and to the public challenge report: the qualitative reports collected from Professional Hackers were subjected to open coding and conceptualization, with the aim of extracting a taxonomy of attack tasks and a set of models of the hackers' behaviour. The qualitative report collected from the Public Challenge winner has been annotated with concepts taken from the taxonomy built in the first experiment (closed coding) and when these were not sufficient, the taxonomy has been extended with new concepts that emerged in the second experiment (open coding). The second experiment served to confirm the taxonomy by adopting it in a new context, as well as an extension of the initial taxonomy with concepts associated to the behaviours of a different category of hackers: those participating in the public challenge, who are typically non-professional hackers.

2.1 Data Collection from Professional Hackers

The three industrial project partners, Nagravision, SafeNet and Gemalto, are world market leaders in their digital security fields. They developed the case study software, and in particular the client-side Android apps of which the security-sensitive parts were implemented in native dynamically linked libraries that were protected by means of the protection tool chain produced by the Aspire project. DemoPlayer is a media player provided by Nagravision. It incorporates DRM (Digital Right Management) that needs to be protected. LicenseManager is a software license manager provided by SafeNet. OTP is a one time password authentication server and client provided by Gemalto. Table 2 (top) shows the lines of code (measured by `sloccount` (Wheeler 2001)) of the three industrial case study applications. For each case study (first column), the table reports the amount of C code (in "*.c" and "*.h" files, respectively), the Java code (in "*.java" files) and the C++ code (in "*.ccp" and "*.c++" files). Each application was protected by the configuration of protections that was deemed most effective in each specific case by the corresponding company's security experts, based on the security requirements of their specific industrial use case. Table 1 lists the deployed protections.

The professional penetration testers involved in the industrial case studies work for security companies that offer third party security testing services. The industrial partners of the project resort routinely to such companies for the security assessment of their products. Such assessments are carried out by hackers with substantial experience in the field, equipped with state-of-the-art tools for reverse engineering, static analysis, debugging, tracing and profiling, etc. Moreover, these hackers are able to customize existing tools, to develop and add plug-ins to existing tools, as well as to develop new tools if needed. In our case, external hacker teams have been augmented/complemented with/by internal professional hacker teams, consisting of security analysts employed by the project's industrial partners. The exact composition of the hacker teams, such as the name and the number of the hacker participants could not be disclosed for confidentiality reasons.

The task for the hacker teams consisted of obtaining access to some sensitive assets secured by the protections. Specifically, the task for the DemoPlayer application was to

Table 1 Protections applied to each industrial use case

Industrial UC	Data obfusc	Anti debug	Remote attestation	Code mobility	Client-server splitting	Virtualization obfusc	WBC	Binary obfusc	Diversified crypto libs
DemoPlayer	×	×		×		×	×		
LicenseManager	×		×		×	×	×	×	
OTP		×						×	×

Table 2 Size of industrial case study applications and the Public Challenges applications in SLoC per file type, before the protection tool chain is applied

Application	C	H	Java	C++	Total
DemoPlayer	2,595	644	1,859	1,389	6,487
LicenseManager	53,065	6,748	819	–	58,283
OTP	284,319	44,152	7,892	2,694	338,103
Challenge 1	134	8	–	–	142
Challenge 2	86	8	–	–	94
Challenge 3	90	8	–	–	98
Challenge 4	118	8	–	–	126
Challenge 5	109	8	–	–	117
Challenge 6	201	8	–	–	209
Challenge 7	156	8	–	–	164
Challenge 8	70	8	–	–	78

violate a specific DRM protection; for LicenseManager it was to forge a valid license key; for OTP it was to successfully generate valid one time passwords without valid credentials. For confidentiality reasons on the industrial use cases, programs could not be shared among different companies and each hacker team only attacked the program owned by the corresponding company.

The hacker team activities could not be traced automatically or through questionnaires. In fact, such teams ask for minimal intrusion into the daily activities performed by their hackers and are only available to report their work in the form of a final, narrative report. For instance, it was not acceptable for security companies to video record industrial hackers while working, or to use screen capturing tools or other intrusive measures to have additional information to compare the results from the reports to.

As a consequence, we had no choice but to adopt a qualitative analysis method. Based on existing qualitative research techniques (Flick 2009), we defined the qualitative analysis method to be adopted in our study, consisting of the following phases: (1) data collection; (2) open coding; (3) conceptualization; (4) model analysis. Although some of the practices that we have adopted are in common with grounded theory (GT) (Glaser and Strauss 1967; Strauss and Corbin 1990), the following key practices of GT (Stol et al. 2016) could not be applied: immediate and continuous data analysis, theoretical sampling, and theoretical saturation, because we had no option to continue data sampling based on gaps in the inferred theory.

Although the final hacker reports are in a free format, we wanted to make sure that some key information was included, in particular information that can provide clues about the ongoing program comprehension process. Hence, before the involved professional hackers started their task, we shared with them a report template where we have asked them to cover the following points in their final attack report:

1. type of activities carried out during the attack: detailed indications about the type of activities carried out to perform the attack and the proportion of time devoted to each activity.
2. level of expertise required for each activity: which of the successful actions required a lot of expertise, which could be done easily.

3. encountered obstacles: detailed description of the obstacles encountered during the attack attempts. In particular, hackers were asked to report any software protection that they think was put into place to prevent the attack and that actually represented a major obstacle for their work.
4. decisions made, assumptions, and attack strategies: description of the attack strategy and how it was adjusted whenever it proved ineffective. Hackers were asked to describe the initial attempts and the decisions (if any) to change the strategy and to try alternative approaches.
5. exploitation on a large scale in the real world: for attacks that succeeded once in the lab, attackers were asked to describe what work would be required to exploit them in the real world (i.e., on a large scale, on software running on standard devices instead of on lab infrastructure, with other keys, etc.).
6. return / remuneration of the attack effort: quantification of the attack effort, if possible economically, so as to provide an estimate of the kind of remuneration that would justify the amount of work done to carry out the attack.

In particular, the second point refers to the *level* of expertise required by the task, e.g., running an existing attack tool does not require a major expertise, while interpreting complex tool output or customizing/extending the tool requires substantial higher expertise.

Each industrial case study was available to hacker teams for a period of 30 days. During that period they worked full-time on it. At the end of this period, hacker teams delivered their reports. While, in general, attack reports covered the above points, not all of the points are necessarily covered in all attack reports or with the same level of details. In particular, quantitative data, such as the proportion of time devoted to each activity, were never provided, whereas qualitative indications about several of the suggested dimensions are present in all reports, though with different levels of verbosity and detail. The reports themselves cannot be made public because of confidentiality requirements. Only the academic Aspire project partners had access to all three reports.

2.2 Data Collection from Public Challenge Winner

The Public Challenge consisted of a set of eight small applications, written by Aspire project members and protected by eight distinct combinations of Aspire protections. Smaller applications were more appropriate for a public challenge, because we expected mostly practitioners to participate, rather than professional hackers.

The eight applications, hereafter called challenges, have been made available on-line to registered users who could download and attack them. Registration could be done anonymously, as only a valid email address had to be provided. The first three participants to break a challenge had their self-chosen user names listed next to that challenge on the challenge web site, together with the timestamp on which the correct solution was submitted. Furthermore, the first successful attacker of each challenge was rewarded with a monetary prize of €200. To be eligible for this prize, a participant had to agree to participate (through e-mail) in a post-mortem forensic interview, in which she/he has been asked to describe how the challenge was attacked and broken, which tools were used, what the general attack strategy was, etc. Furthermore, she/he had to reveal her/his real identity to the organizers of the challenge to enable the pay out of the prize. By implementing the Public Challenge in this way, we again aimed for complementing the data collection of the industrial use cases. Whereas the industrial hackers are white-hat professionals, the targeted audience of this Public Challenge and its prize money was the amateur, hobby, and black-hat community. Also in this case, collecting data from multiple types of attackers helps towards building more generally applicable models.

We provided attackers with GNU/Linux binaries of all challenges in addition to Android binaries, Android being the main demonstration platform of Aspire. By letting the participants choose, hackers and reverse engineers who are skilled in GNU/Linux have not been put off by the prospect of having to port their hacking tools and environment to an Android environment.

The asset to be protected in all the applications is a random string of 64 characters in length, which we will refer to as the *key*. Users received individualized versions of the application, with every generated binary containing a different key, and could request up to three different instances of every challenge. Each instance has a unique key, and the deployed protections are invoked with unique random seeds[2] where applicable, such that code looks different for each instance. As valid keys can contain both upper and lower case characters and numbers, the search space is large enough that attackers are not able to brute force the correct key. The command-line application checks its first argument against this key, and prints out whether or not the argument matches the key. Thus upon entering the correct key as input, it becomes clear if the attack was successful or not. In seven challenges, the answer appears immediately, in one challenge it appears after days only. The latter case was engineered to force attackers to tamper with the code, such that the anti-tampering protections could be evaluated. The attackers could then submit a combination of the identifier of the attacked binary and the key as a solution to the challenge.

Table 3 provides an overview of deployed protections. Besides the indicated protections, all challenges are protected with randomized control-flow obfuscations (opaque predicates, branch functions and function flattening), anti-callback stack checks, and offline code guards. Table 2 (bottom) shows the eight applications' source code sizes. Whereas they are small in that respect, the protected binaries were much larger than these source sizes hint at, because those binaries contain protection functionality on top of the protected application code. The code sections in the binaries, from which attackers start their attack, range from approximately 12kb to 5MB. As all code in those sections is mingled by default by the Aspire protection tool chain, the attackers cannot single out the application code from the protection code easily.

Even though the asset is the same in all challenges (i.e., a key with the same function; obviously the key value differs from binary to binary), and the program I/O behaviour is the same in all challenges, we have decided to tailor the source code of each challenge to suit the protections that are applied in each of those challenges. We did so because different protection combinations can trigger different attack paths, and because different code features lend themselves to different forms of protections. Also in this way, we aim to broaden our data collection in order to yield more generally applicable models, in this case by ensuring that different attack paths are included in the data collection.

[2] The random seed was not meant to decide what protection to deploy or in what variant. The random seed is used by protections to initialize values, e.g., the value to use as key, and to diversify the way certain protections are injected and obfuscated, such that the injected code cannot be identified through trivial pattern matching. Nonetheless, we verified that the randomization process did not change the code and execution patterns in such a way that diversified versions required different ways to be attacked, as it would have altered the analysis of the results.

For each of the challenges summarized in Table 3, the source code characteristics and the challenge-specific protections are the following ones:

Ch 1: In this challenge, an array is constructed by a block of code that is protected with an anti-debugging protection that prevents an attacker from attaching his own debugger to study the internal operation of the program (Abrath et al. 2016). Next, code mobility (Cabutto et al. 2015) is used to protect the XORing of this array with mobile data (i.e., data that is not present in the static program binary, but is downloaded from a secure server at run time). Mobile code also contains the invocation to the `strncmp` C function that checks the validity of the key, this hiding that operation.

Ch 2: The key is split into 16 integers, each of which is encoded using Residue Number Coding (RNC) (Demissie et al. 2015). The input to the program is also split into 16 integers, and these integers are then compared with the RNC-encoded key.

Ch 3: This challenge uses RNC data obfuscation and anti-debugging.

Ch 4: The main function is protected by moving it into a virtual machine (VM): the main function body is translated into bytecode that is interpreted by the VM, instead of being executed directly.

Ch 5: The key is used as ciphertext for White Box Cryptography (WBC) with a fixed key (Wyseur 2009). This is then "decrypted" when a challenge instance is generated, and the decrypted key is stored in the challenge instance. Furthermore, the WBC code has been protected with anti-debugging.

Ch 6: The key is checked in two parts: the first half of the key is checked byte per byte. The code that checks the key performs a very long delay loop (two nested loops, both of which only finish after looping through 2^{64} values) and a sleep of about 11 days after each character is checked. These delays are to encourage the attackers to try to modify the binary to remove or shorten the delay to verify the key, i.e., to tamper with the code, which is protected with Remote Attestation (RA) (Collberg and Nagra 2009). The next part of the binary is protected using mobile code, so that if the RA component detects tampering, the protection server can trigger a tamper response in the code mobility component.

Ch 7: This challenge starts from code that has "ugly" control flow at the source level. Then this code has binary control flow obfuscations applied, and is furthermore protected with anti-debugging.

Table 3 Protections applied to each challenge (Ch)

Ch	Data obfusc	Anti debug	Remote attestation	Code mobility	Client-server splitting	Virtualization obfusc	WBC
1		×	×	×			
2	×						
3	×	×					
4		×				×	
5		×					×
6			×	×			
7		×					
8		×			×		

Ch 8: The code is protected with client-server code splitting (Ceccato et al. 2007), where each character is sent individually to the server, and the client only asks for the correctness of the next key character if the previous one was correct.

The Public Challenge was advertised on the reverse engineering sub-reddit, on Twitter, in the institutions that partnered in the Aspire project, and in summer schools to which they participated.

Five challenges were broken (in the order 5, 7, 2, 3, and 4) by the same hacker, who participated in the post-mortem forensic interview. The content of the interview represents the qualitative report, comprising several paragraphs dedicated to each one of the five broken challenges as well as some general considerations and impressions on the challenges, e.g., on how to prepare the attacks and activities shared among challenges. After an initial email, asking for general explanations about the successful attacks ("how did you find the keys?", "which tools did you use?", "how did you approach the different challenges?", "which difficulties (if any) did you experience and how did you tackle them?"), the interview proceeded with further requests of clarifications and explanations, based on the hacker's replies. Examples of follow-up questions are: "Could you estimate how much effort / time you spent per challenge?", "How did you know it was WBC?", "For the seventh and third challenges, apart from the checksums, did you experience any problems when attaching the debugger? If so, how did you circumvent them; if not, which debugger did you use and how did you attach to the program? (Did you notice that there were two processes running?)", "How did you know where to put the breakpoints?". The interview was closed when the associated report contained enough information to understand the attack strategy as well as the technical details behind the five successful attacks.

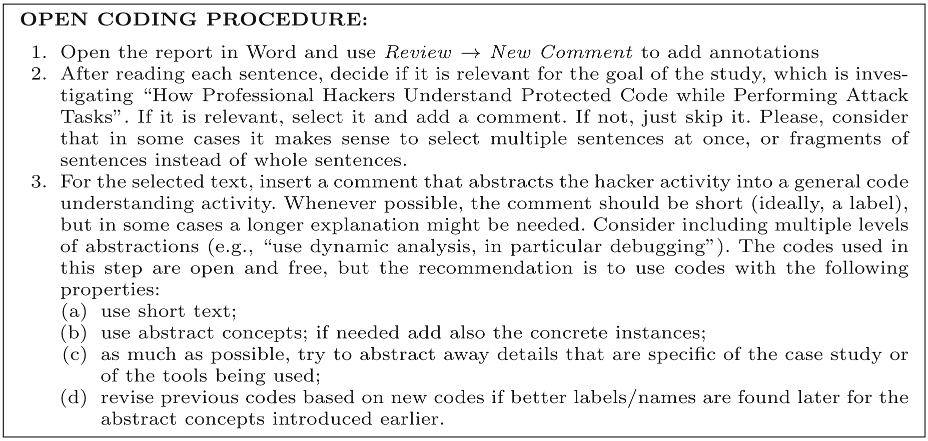
The interview also confirmed that the participant who solved the public challenges was indeed an experienced amateur hacker, with access to advanced, but publicly available tools. This participant, thus, fitted the intended target of the Public Challenge. Considering that this profile differs from the professional hackers involved in the first experiment, we aim for a substantial extension of the initial taxonomy built just on industrial settings, by including data from two diverse settings.

2.3 Open Coding and Conceptualization (Professional Hacker Experiment)

Open coding was applied to the qualitative data collected from the Professional Hackers. Open coding of the reports was carried out by each academic institution participating in the Aspire project. Coding by seven different coders was conducted autonomously and independently. Only high level instructions have been shared among coders before starting the coding activity, so as to leave maximum freedom to coders and to avoid the introduction of any bias during coding. These general instructions are reported in Fig. 1.

The annotated reports obtained after open coding were merged into a single report containing all collected annotations. We have not attempted to unify the various annotations because we wanted to preserve the viewpoint diversity associated with the involvement of multiple coders operating independently from each other. Unification is one of the main goals of the next phase, conceptualization.

Conceptualization was applied to the annotated data collected from the Professional Hackers. This phase consists of a manual model inference process carried out jointly by all coders. The process involves two steps: (1) concept identification; (2) model inference. The goal of *concept identification* is to identify key concepts that coders used in their annotations, to provide a unique label and meaning to such concepts and to organize them into a



OPEN CODING PROCEDURE:

1. Open the report in Word and use *Review* → *New Comment* to add annotations
2. After reading each sentence, decide if it is relevant for the goal of the study, which is investigating "How Professional Hackers Understand Protected Code while Performing Attack Tasks". If it is relevant, select it and add a comment. If not, just skip it. Please, consider that in some cases it makes sense to select multiple sentences at once, or fragments of sentences instead of whole sentences.
3. For the selected text, insert a comment that abstracts the hacker activity into a general code understanding activity. Whenever possible, the comment should be short (ideally, a label), but in some cases a longer explanation might be needed. Consider including multiple levels of abstractions (e.g., "use dynamic analysis, in particular debugging"). The codes used in this step are open and free, but the recommendation is to use codes with the following properties:
   (a) use short text;
   (b) use abstract concepts; if needed add also the concrete instances;
   (c) as much as possible, try to abstract away details that are specific of the case study or of the tools being used;
   (d) revise previous codes based on new codes if better labels/names are found later for the abstract concepts introduced earlier.

Fig. 1 Coding instructions shared among coders of the Professional Hackers report

concept hierarchy. The most important relation identified in this step is the "is-a" relation between concepts, but other relations, such as aggregation or delegation, might emerge as well. In this step, the main focus is a static, structural view of the concepts that emerge from the annotations. The output is thus a so-called "lightweight" ontology (i.e., an ontology where the structure is modelled explicitly, while axioms and logical constraints are ignored).

The goal of *model inference* is to obtain a model with explanation and predictive power. To this aim, the concepts obtained in the previous step are revised and the following relations between pairs of concepts are conjectured, based on considerations or observations formulated by the participants: (1) temporal relations (e.g., *before*); (2) causal relations (e.g., *cause*); (3) conditional relations (e.g., *condition for*); (4) instrumental relation (e.g., *used to*). Evidence is sought for such conjectures in the annotations. It should be noted that relations are not based on the exact words used in the reports. Concept relations are rather based on the meaning of participants comments, so that a consistent model is inferred even if different participants used different words to mean the same relation. The outcome of this step is a model that typically includes a causal graph view, where edges represent causal, conditional and instrumental relations, and/or a process view, where activities are organized temporally into a graph whose edges represent temporal precedence. This step is deemed concluded when the inferred model is rich enough to explain all the observations encoded in the annotations of the hacker reports, as well as to predict the expected hacker behaviour in a specific attack context, which depends on context factors such as the features of the protected application, the applied protections, the assets being protected, the expected obstacles to hacking.

Correspondingly, two joint meetings (over conference calls) have been organized to carry out the two steps. During each meeting, the report with the merged codes was read sentence by sentence and annotation by annotation. During such reading, abstractions have been proposed by coders either for concept identification (step 1) or for model inference (step 2). The proposed abstractions have been discussed; the discussion proceeded until consensus was reached. During the process, whenever new abstractions were proposed and discussed, the abstractions introduced earlier were possibly revised and aligned with the newly introduced abstractions.

Although the conceptualization phase is intrinsically subjective, subjectivity was reduced by: (1) involving multiple coders with different backgrounds and asking them to reach consensus on the abstractions that emerged from codes; (2) keeping traceability links between abstractions and annotations. Traceability links are particularly important, since they provide the empirical evidence for the inference of a given concept or relation. Availability of such traceability links allows coders to revise their decisions later, at any point in time, and allows external inspectors of the model to understand (and possibly revise/change) the connection between abstractions and initial annotations.

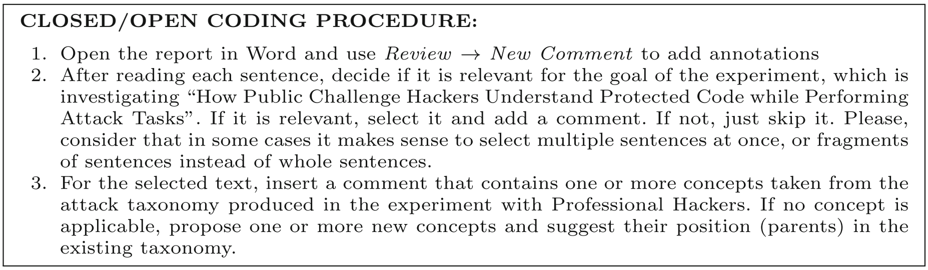
2.4 Taxonomy Extension (Public Challenge Experiment)

The taxonomy extracted from the experiment with Professional Hackers was reused to annotate the report collected from the Public Challenge. It has been extended with new concepts when the existing ones were insufficient for the annotation of the report. The closed/open coding procedure followed to annotate the Public Challenge report is described in Fig. 2. Whenever a relevant text fragment could not be annotated by reusing an existing concept (closed coding), the annotator could recommend a new concept for the given text fragment (open coding).

The annotated reports obtained after individual closed/open coding were merged into a single report containing all collected annotations. Two consensus meetings have been carried out over conference calls to reach a consensus on a unified annotation of the Public Challenge report. During each consensus meeting, the report with the merged codes was read sentence by sentence and annotation by annotation. The proposed annotations with existing or new concepts have been discussed and the discussion proceeded until consensus was reached. During the process, whenever new concepts were approved, their position in the existing taxonomy was also discussed and decided as well as their relations with other concepts in the taxonomy.

3 Results

Table 4 shows the number of annotations produced for the Professional Hacker reports (top) and the Public Challenge winner report (bottom). The three case study reports (indicated as P: DemoPlayer; L: LicenseManager; O: OTP) produced by Professional Hackers have been annotated by seven annotators (indicated as A, B, C, D, E, F, G) from the academic partners of Aspire. Each annotation is labelled by a unique identifier having the following structure: [<*case study*> : <*annotator*> : <*number*>] (e.g., [P:D:7]) to simplify traceability between



CLOSED/OPEN CODING PROCEDURE:

1. Open the report in Word and use *Review* → *New Comment* to add annotations
2. After reading each sentence, decide if it is relevant for the goal of the experiment, which is investigating "How Public Challenge Hackers Understand Protected Code while Performing Attack Tasks". If it is relevant, select it and add a comment. If not, just skip it. Please, consider that in some cases it makes sense to select multiple sentences at once, or fragments of sentences instead of whole sentences.
3. For the selected text, insert a comment that contains one or more concepts taken from the attack taxonomy produced in the experiment with Professional Hackers. If no concept is applicable, propose one or more new concepts and suggest their position (parents) in the existing taxonomy.

Fig. 2 Coding instructions shared among coders of the Public Challenge report

Table 4 Number of annotations by annotator and by case study report

Industrial	Annotator							
Case study	A	B	C	D	E	F	G	Total
P	52	34	48	53	43	49	NA	279
L	20	10	6	12	7	18	9	82
O	12	22	NA	29	24	11	NA	98
Total	84	66	54	94	74	78	9	459

Public challenge	Annotator team				
	T1(A,G)	T2(C,D)	T3(B,F)	T4(E)	Total
C2	11	14	4	5	34
C3	3	9	2	3	17
C4	21	44	12	7	84
C5	10	12	3	3	28
C7	3	4	3	1	11
Common	22	46	9	14	91
Total	70	129	33	33	265

inferred concepts and models on one side and annotations supporting them on the other side. The reports have been processed in the same order by all the annotators, that is P, L and O. The reports on the five broken challenges produced from the interview with the Public Challenge winner have been annotated by four teams of annotators (indicated as T1, T2, T3, T4), one from each academic institution participating in the Aspire project. The composition of annotator teams is reported in parentheses near to the team name.

In total, 459 annotations have been produced for the Professional Hacker reports. Case study P received considerably more annotations than L and O, mostly because of the amount and richness of the information available in this hacker report. The number of annotations made by different annotators is quite consistent across the case studies (NA = *Not Available* indicates that the annotator did not annotate that specific report), with L showing the highest variability (min = 6; max = 20). For the Public Challenge report, 265 annotations have been produced in total by the four teams, with wide variability (min = 33; max = 129) indicating a spectrum of attitudes on the amount of concepts used for each text fragment to be annotated. These 265 annotations clearly form a relevant extension over the 459 annotations considered in our previous work (Ceccato et al. 2017).

The Public Challenge annotations have been applied to both text describing a specific challenge and text describing general considerations concerning all the challenges. General considerations cover aspects that are independent of the single challenge, like environment preparation and use of analysis tools. Table 4 reports the number of annotations associated to text of the report that was explicitly referred to a specific challenge. It is possible to see how text concerning individual challenges has been heavily annotated. The most complicated challenge to break, i.e., the fourth challenge that used white box crypto, deserved a long description by the hacker and it was annotated by teams with a number of annotations similar to that of the L and O industrial case studies.

Annotators were the same for the two experiments. The difference in the annotation procedure between the first and the second experiment (i.e., individual annotators versus teams of annotators) is due to the slightly different settings between the two experiments. In fact, in the second experiment annotators started from the existing taxonomy, not from scratch, and they could benefit from the experience maturated in annotating the first experiment reports. Teams were used in the second experiment to anticipate some discussion within each team when annotating the reports, to allow for an easier successive plenary meeting in which consensus was aimed for.

3.1 Identified Concepts

Figures 3 and 4, and 5 show the taxonomy of concepts resulting from the conceptualization process carried out by the annotators. New concepts introduced in the Public Challenge experiment are underlined; concepts emerged in both experiments are in boldface.

The top concepts in the taxonomy correspond to the main notions that are useful to describe the hacker activities. These are: *Obstacle, Analysis / reverse engineering, Attack strategy, Attack step, Workaround, Weakness, Asset, Background knowledge, Tool, Software elements, Difficulty* (the last one, *Difficulty*, was introduced only when the Public Challenge data have been processed). The taxonomy in OWL format is available online at: http://selab.fbk.eu/ceccato/hacker-study/EMSE2017.owl. To help readability, in the graphical representation of our taxonomy, concepts are ordered according to possible phases, e.g the preparation comes ideally first, then some code understanding is done, the attack attempt takes place before the evaluation of the attack result. We did not opt for a more formal ordering, such as the number of concepts or the size of reports, because it would have been arbitrary or based on sensitive information that we could not disclose. Moreover, it would be difficult to keep a consistent ordering approach in future experiments, possibly conducted by other researchers, where data sharing might be still difficult because of confidentiality issues.

3.1.1 Obstacle

As expected, in the *Obstacle* hierarchy (Fig. 3) we find the protections that are applied to the software to prevent the hacker attacks (under concept *Protection*). We observe that this is not the only kind of obstacle reported by hackers.

In particular, the *Execution environment* may also be a major impediment to the completion of an attack. In a report we read *"Aside from the [omissis]*[3] *added inconveniences [due to protections], execution environment requirements can also make an attacker's task much more difficult. [omissis] Things such as limitations on network access and maximum file size limitations caused problems during this exercise"*; on this part one coder annotated [P:F:7]: "General obstacle to understanding [by dynamic analysis]: execution environment (Android: limitations on network access and maximum file size)".

3.1.2 Difficulty

Most difficulties (see *Difficulty* hierarchy in Fig. 3) are problems encountered by the hacker while performing the Public Challenge, due to lack of knowledge (*"This was my first major encounter with ARM arch and Linux platform"*) or portability (*"The problem as far as I could gather was inside `libwebsockets` processing, probably because of my device*

[3] With the placeholder "[omissis]" we indicate that a part of the text is not reported either because it cannot be disclosed for confidentiality reasons or because it is not relevant since we want the reader to focus on the most meaningful (and shorter) portion.

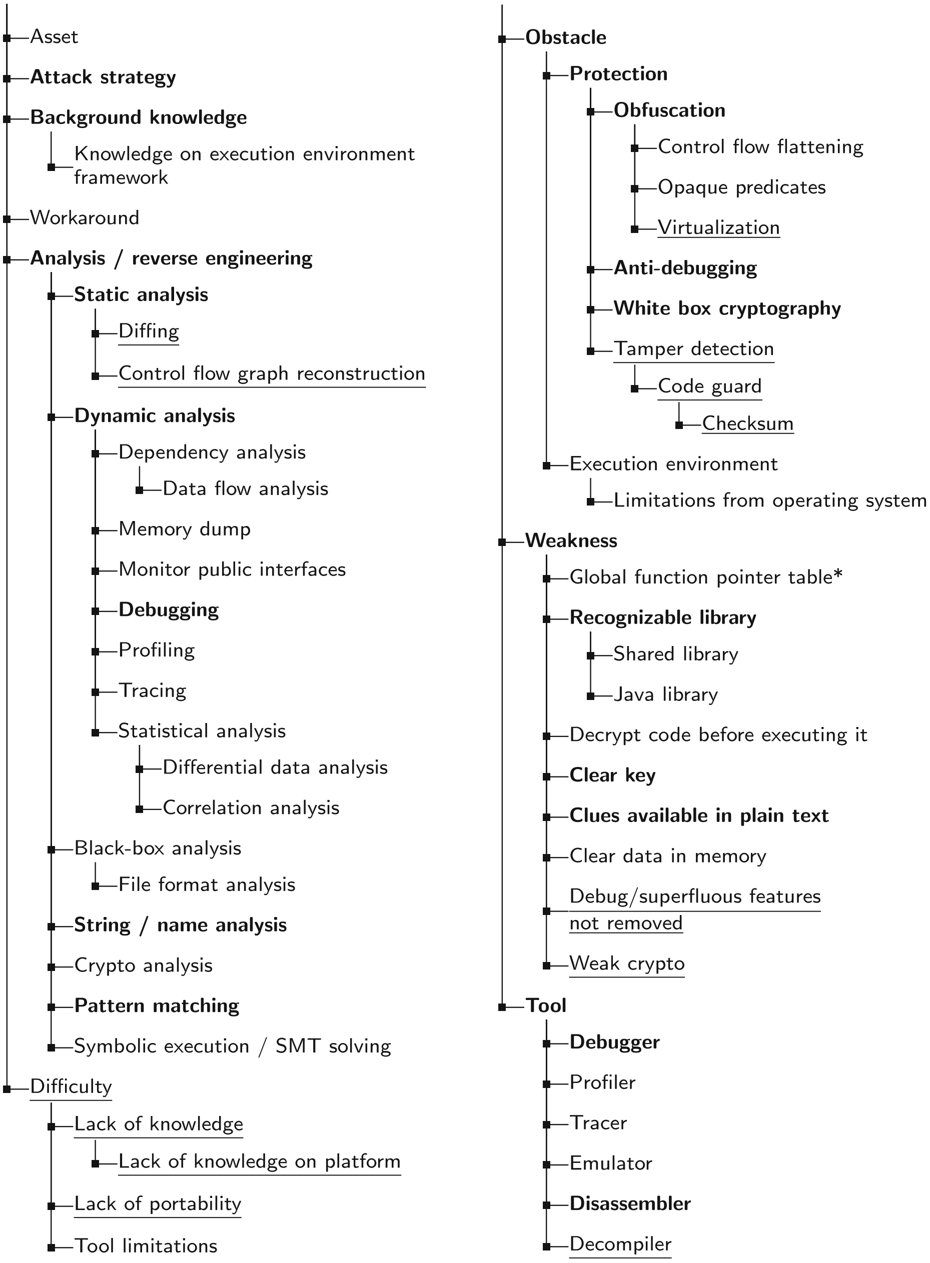


Fig. 3 Taxonomy of extracted concepts (part I): the analysis methods and tools hackers may use (*Analysis / reverse engineering*, *Tool*), weaknesses in design and coding of the application to protect that may help the hacker tasks (*Weakness*), the difficulty hackers may experience when trying to perform an attack task (*Difficulty*), the protections a defender can place to limit certain attack steps (*Obstacle*), and other high-level concepts that characterize the hacking scenarios (*Asset*, *Attack strategy*, *Background knowledge*, *Workaround*). * indicates multiple inheritance; new concepts added during the second qualitative experiment are underlined; concepts emerged in both experiments are in **boldface**

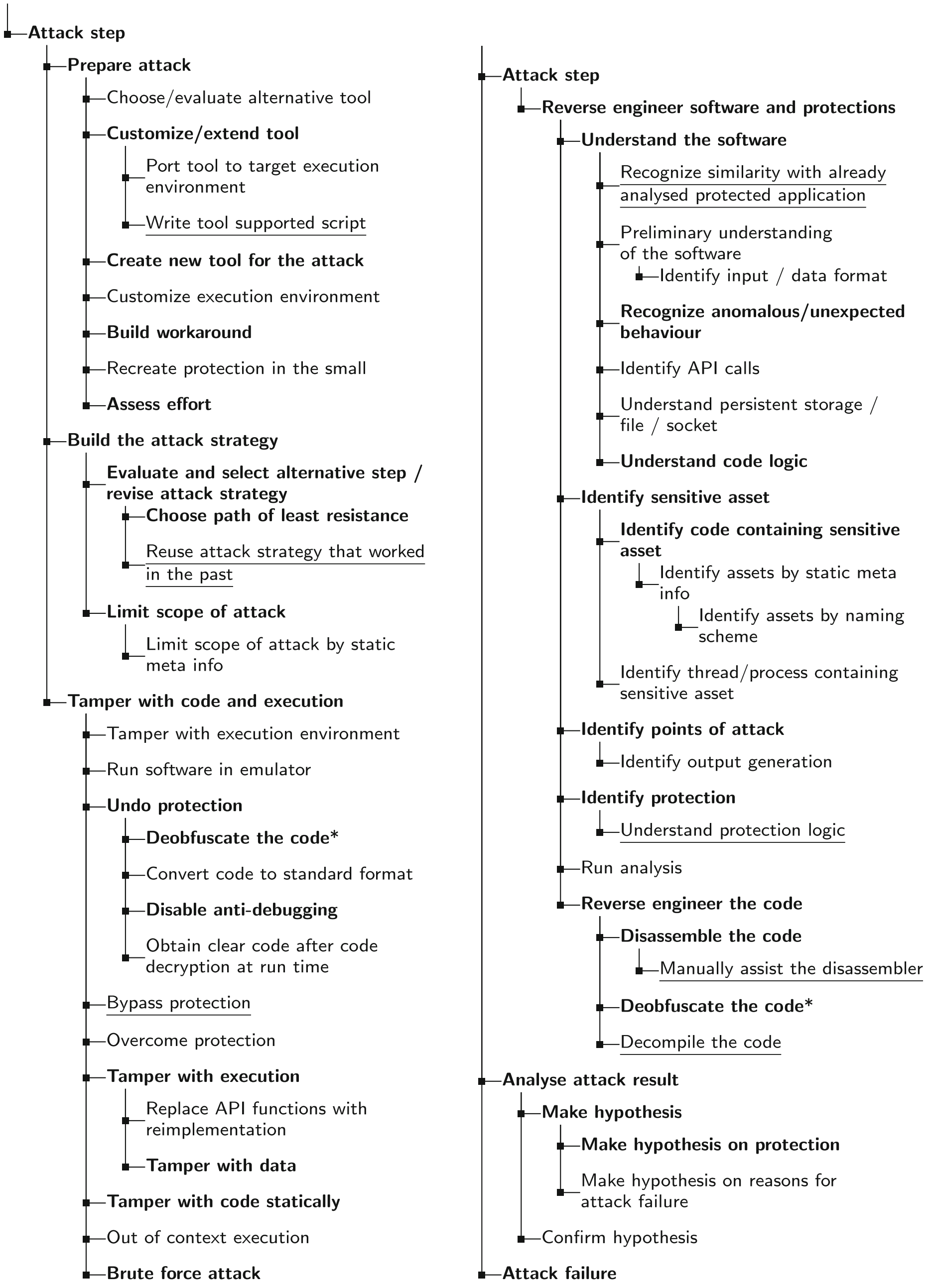

Attack step
Prepare attack
Choose/evaluate alternative tool
Customize/extend tool
Port tool to target execution environment
Write tool supported script
Create new tool for the attack
Customize execution environment
Build workaround
Recreate protection in the small
Assess effort
Build the attack strategy
Evaluate and select alternative step / revise attack strategy
Choose path of least resistance
Reuse attack strategy that worked in the past
Limit scope of attack
Limit scope of attack by static meta info
Tamper with code and execution
Tamper with execution environment
Run software in emulator
Undo protection
Deobfuscate the code*
Convert code to standard format
Disable anti-debugging
Obtain clear code after code decryption at run time
Bypass protection
Overcome protection
Tamper with execution
Replace API functions with reimplementation
Tamper with data
Tamper with code statically
Out of context execution
Brute force attack
Attack step
Reverse engineer software and protections
Understand the software
Recognize similarity with already analysed protected application
Preliminary understanding of the software
Identify input / data format
Recognize anomalous/unexpected behaviour
Identify API calls
Understand persistent storage / file / socket
Understand code logic
Identify sensitive asset
Identify code containing sensitive asset
Identify assets by static meta info
Identify assets by naming scheme
Identify thread/process containing sensitive asset
Identify points of attack
Identify output generation
Identify protection
Understand protection logic
Run analysis
Reverse engineer the code
Disassemble the code
Manually assist the disassembler
Deobfuscate the code*
Decompile the code
Analyse attack result
Make hypothesis
Make hypothesis on protection
Make hypothesis on reasons for attack failure
Confirm hypothesis
Attack failure

Fig. 4 Taxonomy of extracted concepts (part II): the attack steps hackers may perform. They include the operations to prepare the attack (*Prepare attack*) and decide how to mount it (*Build the attack strategy*), the tasks to understand the software through reverse engineering the application code (*Reverse engineer software and protection*), the modifications to code and executions to tamper with the application (*Tamper with code and execution*), and the tasks to evaluate whether the attack was successful or not and learn from errors (*Analyse attack result*). * indicates multiple inheritance; new concepts added during the second qualitative experiment are underlined; concepts emerged in both experiments are in **boldface**

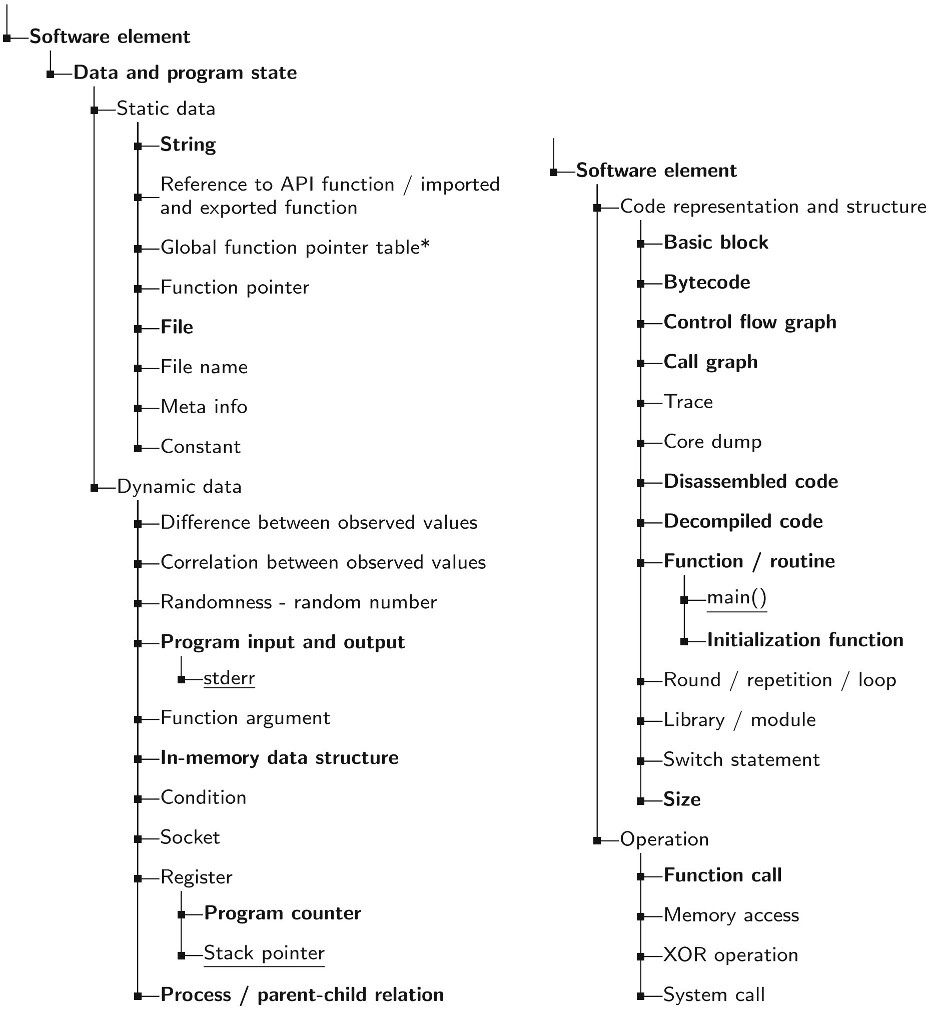


Fig. 5 Taxonomy of extracted concepts (part III): software elements. * indicates multiple inheritance; new concepts added during the second qualitative experiment are underlined; concepts emerged in both experiments are in **boldface**

setup"). Another difficulty (formerly classified as an *Obstacle* (Ceccato et al. 2017)) is represented by *Tool limitations*, for which we can find annotated sentences such as [P:A:33]: "Attack step: overcome limitation of an existing tool by creating an ad hoc communication means" in the professional hacker reports.

3.1.3 Analysis/reverse Engineering

The *Analysis / reverse engineering* hierarchy (see Fig. 3) is quite rich and interesting. It includes advanced techniques that are part of the state of the art of the research in code analysis, such as *Symbolic execution / SMT solving; Dependency analysis; Statistical analysis*. Of course, hackers are well aware of the most recent advances in the field of code analysis.

Control flow reconstruction and *Diffing* were added when processing the report from the Public Challenge. In particular, the Public Challenge hacker compared (using the utility `diff`) the original source code of a publicly available interpreter with the decompiled code obtained from the challenge, to detect any modification that could have been made to that interpreter, that was in fact re-used in the Aspire project to protect the challenge (*"I was cross-referencing the original [omissis] source code with challenge's decompilation to spot the differences in processing [omissis] file"*) .

3.1.4 Attack Step

The central concept that emerged from the hacker reports is *Attack step*, whose hierarchy is shown in Fig. 4. An *Attack step* represents each single activity that must be executed when implementing a chosen *Attack strategy*. The top level concepts under *Attack step* correspond to the major activities carried out by hackers. Hackers that opted for dynamic attack strategies first of all prepare the attack (concept *Prepare attack*) to ensure the code can be executed under their control. Then, they usually spend some time understanding the code and its protections by means of a variety of activities that are sub-concepts of *Reverse engineer software and protections* in the taxonomy. Once they have gained enough knowledge about the software under attack, they build a strategy (concept *Build attack strategy*, they execute any necessary, preliminary task (concept *Prepare attack* and they actually execute the attack by manipulating the software statically or at run time (concept *Tamper with code and execution*. Finally, they analyse the attack results and decide how to proceed (concept *Analyse attack result*).

Reverse Engineer Software and Protections The attack step *Reverse engineer software and protections* includes several activities in common with general program understanding (see Fig. 4), but it also includes some hacking-specific activities. For instance, recognizing the occurrence of program behaviours that are not expected for the software under attack (concept *Recognize anomalous/unexpected behaviour*; [P:A:27] "Identified strange behaviour compared to the expected one (from their background knowledge)") is important, since it may point to computations that are unrelated with the software business logic and are there just to implement some protection. It might also point to variants of well known protections ([P:E:17] "Infer behaviour knowing AES algo details"). Identification of sensitive assets in the software (concept *Identify sensitive assets*; [P:D:4] "prune search space of interesting code, using very basic static (meta-) information") and of points of attack (concept *Identify points of attack*; [P:E:14] "Analyse traces to locate output generation") are other examples of hacking-specific program understanding activities. The *Identify protection* sub-hierarchy is also unique to code hacking and can take advantage of similarities with previously attacked applications or with manual inspection of the protection logic (*"they calculated checksums of a certain sections of binary and compared it with input (which was divided by dwords)"*).

Build Attack Strategy When iteratively building the attack strategy (concept *Build attack strategy*, Fig. 4), it is important to be able to reduce the scope of the attack to a manageable portion of the code. This key activity is expressed through the concept *Limit scope of attack* ([O:D:5] "use symbolic operation to focus search"). Within such narrowed scope, hackers evaluate the alternatives and choose the path of least resistance (concepts *Evaluate and select alternative steps / revise attack strategy* and *Choose path of least resistance*; see, e.g., the sentence: *"As the libraries are obfuscated, static analysis with a tool such as IDA Pro is* *difficult at best"*, annotated as [P:D:5] "discard attack step/paths"). When possible, they try to reuse an attack strategy that worked in the past (*"I used same strategy to extract the key"*).

Tamper with Code and Execution Another remarkable difference from general program understanding is the substantial amount of code and execution manipulation carried out by hackers. Indeed, a core attack step consists of the alteration of the normal flow of execution (concept *Tamper with code and execution* in Fig. 4). This is achieved in many different ways, as apparent from the richness of the hierarchy rooted at *Tamper with code and execution*. Some of them are hacking-specific and reveal a lot about the typical attack patterns. For instance, activity *Replace API functions with reimplementation* is carried out to work around a protection, by replacing its implementation with a fake implementation by the hackers ([P:F:49] "New attack strategy based on protection API analysis: replace API functions with a custom reimplementation to be done within the debugging tool"). Activity *Tamper with data* is carried out to alter the program state to defeat a protection (sentence *"to set a fake value in virtual CPU registers in order to deceive the debugged application"*, annotated as [O:D:11] "tamper with data to circumvent triggering protection"). *Out of context execution* is carried out to run the code being targeted by an attack, e.g., a protected function, in isolation, as part of a manually crafted `main` program (sentence *"write own loader for [omissis] library"*, annotated as [L:D:20] "adapt and create environment in which to execute targeted code out of context"). Moreover, hackers tamper with the execution to undo the effects of a protection (concept *Undo protection*), often to reverse engineer the clear code from the obfuscated one (concepts *Deobfuscate the code*, *Convert code to standard format*, and *Obtain clear code after code decryption at run time*). When possible, they prefer to bypass a protection rather than undoing it (concept *Bypass protection* used to annotate the following sentences from the Public Challenge report: *"I was successful in overriding checksum mechanism"*; *"I [omissis] put breakpoints after they executed so I simply caught expected results"*).

3.1.5 Software Elements

To put the concepts that describe the program comprehension processes and activities in perspective, it is interesting to know the artifacts on which attackers focus in programs, i.e., the structures, components and elements they consider as relevant program aspects to interpret and manipulate in their reverse-engineering and tampering attacks. To that extent, we identified all the *Software elements* that the attackers mentioned in their reports and interview responses. Figure 5 provides a taxonomy of the extracted terms, with four top-level concepts.

The concept *Code representation and structure* contains the forms and representations of software that attackers consider, varying from very concrete (e.g., *Core dump*) to more abstract (*Decompiled code*), from static (*Control flow graph* and *Call Graph*) to dynamic (*Trace*), and the structural elements they consider at those different levels and in the different forms. The feature *Size* plays an important role, because attackers consider the apparent size of different components (ranging from the whole application under attack to the size of individual functions) for deciding on the best attack strategy.

Semantics of the code structures in the representations is often derived from, and attached to, specific operations. Four relevant ones are covered by the Operation concept. Some are generic, like *Function call*, *Memory access* and *System call*. *XOR operation*s are considered specifically in the context of cryptographic primitives and attacks thereon.

While performing dynamic attacks steps, e.g., by observing program execution with a debugger or by analysing execution traces, the attackers observe and track different aspects and features of the dynamic program data and states. Those are covered in the concept *Data and program state*. Most are generic because they are relevant in almost all programs; other concepts, such as those evolving around differences, correlation, and randomness are again related to cryptography, where those aspects are understood to relate to weaknesses, e.g., with respect to side-channel attacks.

Finally, the concept *Static data* covers all forms of extra information (i.e., not just the actual static code and data that is part of the running program) that attackers can extract from the statically available attack objects, i.e., from the executable files.

3.2 Comparison Between Professional Hackers and Public Challenge Winner

Here, we present the differences that we spotted in the concepts identified between the two experiments. As such, this difference should not be interpreted as representative of the general difference between public challenges and professional industrial hackers.

Table 5 shows the taxonomy concepts that are used to annotate uniquely the professional hacker reports, uniquely the public challenge report, or both. The number of concepts is computed by simply counting the leaves in the taxonomy. The numbers indicate a substantial degree of taxonomy reuse in the second experiment (41%), despite the completely different setting, which involved small, ad-hoc programs instead of large, industrial applications, and an amateur hacker versus professionals. This means the original ontology, emerged from the first experiment (Ceccato et al. 2017), included part of the core concepts required to annotate hacker reports. The second experiment was effective in adding new concepts, missing from the original ontology.

Figures 3, 4 and 5 depict in boldface the 53 concepts common to the two experiments. All top level concepts (except *Difficulty*, which was added in the second experiment as a similar but distinct *Obstacle*) are used to annotate both experiments. This means that several high level categories of activities were identified in both experiments, including *Analysis / reverse engineering*, *Attack strategy*, *Attack step*, *Background knowledge*, *Tool*. Looking deeper into the taxonomy, we can notice that several nested concepts, including several leaf concepts, are shared between the two experiments. *String / name analysis*, *Pattern matching*, *Dynamic analysis* and in particular *Debugging* are core techniques for *Analysis / reverse engineering* that were used in both experiments.

Among the *Attack step*'s (see Fig. 4) sub-concepts, most of the first and second level ones are found in both experiments. Hackers in both experiments performed activities like: *Reverse engineer software and protections*, *Understand the software*, *Identify sensitive asset*, *Identify points of attack*, *Identify protection*, *Reverse engineer the code*, *Prepare attack*, *Customize/extend tool*, *Create new tool for the attack*, *Build workaround*, *Assess effort*, *Tamper with code and execution*, *Undo protection*, *Tamper with execution*, *Tamper with code statically*, *Brute force attack*, *Build the attack strategy*, *Evaluate and select alternative step / revise attack strategy*, *Limit scope of attack*, *Analyse attack result*, *Make hypothesis*.

Among the deeply nested concepts shared between the two experiments, it is interesting to notice the importance of basic, low level binary code analysis techniques (concepts *Disassemble the code*, *Deobfuscate the code*), which are standard toolkit parts in any successful hacker attack. The concept *Tamper with data* is also interesting. In fact, looking at the text fragments annotated with this concepts, a clear pattern of tampering with the data emerges: hackers set a breakpoint in the binary code and before resuming the execution they alter

Table 5 Comparison of concepts emerged in the two experiments

Unique to professional hackers	76 concepts (45%)
Asset, Black-box analysis, Clear data in memory, Code representation and structure, Condition, Confirm hypothesis, Constant, Control flow flattening, Convert code to standard format, Core dump, Correlation analysis, Correlation between observed values, Crypto analysis, Customize execution environment, Data flow analysis, Decrypt code before executing it, Dependency analysis, Difference between observed values, Differential data analysis, Disassembler, Dynamic data, Emulator, Execution environment, File format analysis, File name, Function argument, Function pointer, Global function pointer table, Identify API calls, Identify assets by naming scheme, Identify assets by static meta info, Identify input / data format, Identify output generation, Identify thread/process containing sensitive asset, Java library, Knowledge on execution environment framework, Library / module, Limit scope of attack by static meta info, Limitations from operating system, Make hypothesis on reasons for attack failure, Memory access, Memory dump, Meta info, Monitor public interfaces, Obtain clear code after code decryption at run time, Opaque predicates, Operation, Out of context execution, Overcome protection, Port tool to target execution environment, Preliminary understanding of the software, Profiler, Profiling, Randomness - random number, Recreate protection in the small, Reference to API function / imported and exported function, Register, Replace API functions with reimplementation, Round / repetition / loop, Run analysis, Run software in emulator, Shared library, Socket, Static data, Statistical analysis, Switch statement, Symbolic execution / SMT solving, System call, Tamper with execution environment, Tool limitations, Trace, Tracer, Tracing, Understand persistent storage / file / socket, Workaround, XOR operation	
Unique to public challenge	23 concepts (14%)
Bypass protection, Checksum, Code guard, Control flow graph reconstruction, Debug/superfluous features not removed, Decompile the code, Decompiler, Difficulty, Diffing, Lack of knowledge, Lack of knowledge on platform, Lack of portability, main(), Manually assist the disassembler, Recognize similarity with already analysed protected application, Reuse attack strategy that worked in the past, Stack pointer, stderr, Tamper detection, Understand protection logic, Virtualization, Weak crypto, Write tool supported script	
Both experiments	70 concepts (41%)
Analyse attack result, Analysis / revrse engineering, Anti-debugging, Assess effort, Attack failure, Attack step, Attack strategy, Background knowledge, Basic block, Brute force attack, Build the attack strategy, Build workaround, Bytecode, Call graph, Choose path of least resistance, Clear key, Clues available in plain text, Control flow graph, Create new tool for the attack, Customize/extend tool, Data and program state, Debugger, Debugging, Decompiled code, Deobfuscate the code, Disable anti-debugging, Disassemble the code, Disassembled code, Disassembler, Dynamic analysis, Evaluate and select alternative step / revise attack strategy, File, Function / routine, Function call, Identify code containing sensitive asset, Identify points of attack, Identify protection, Identify sensitive asset, Initialization function, In-memory data structure, Limit scope of attack, Make hypothesis, Make hypothesis on protection, Obfuscation, Obstacle, Pattern matching, Prepare attack, Process / parent-child relation, Program counter, Program input and output, Protection, Recognizable library, Recognize anomalous/unexpected behaviour, Reverse engineer software and protections, Reverse engineer the code, Size, Software element, Static analysis, String, String / name analysis, Tamper with code and execution, Tamper with code statically, Tamper with data, Tamper with execution, Tool, Understand code logic, Understand the software, Undo protection, Weakness, White box cryptography	
Total	169 concepts

some critical data (the value of a checksum, of a key, etc.) to test some hypothesis or expose some anomalous behaviour. Breakpoint setting and execution state manipulation seems yet another important basic technique that any hacker is willing to use. Hence the importance of *Anti-debugging* protections, which hackers tried to defeat in both experiments (see concept *Disable anti-debugging* in Fig. 4, an *Attack step* common to both experiments). From the point of view of the attack strategy, important low level concepts emerged in both experiments, such as *Make hypothesis on protection* and *Choose path of least resistance*. In the attempt to save and optimize the attack effort, hackers speculate on protections even when they have not much evidence about them and choose the attack path that requires minimum effort based on the hypothesized protection.

Table 5 (top) shows the concepts uniquely used to annotate the professional hacker reports. Some quite advanced analysis and reverse engineering techniques have been used only by professional hackers. For instance: *Symbolic execution / SMT solving*, *Crypto analysis*, *Data flow analysis*, *Differential data analysis*, *Correlation analysis*. The explanation for this might be twofold: on one hand, these are techniques that require specialist competences and dedicated tools, which probably are unavailable to the typically non-professional hacker engaged in the Public Challenge. In our case, indeed, the Public Challenge was solved by a practitioner, whose interest in reverse engineering is purely a hobby. On the other hand, the industrial applications protected in the first experiment were much more complex than the small challenges used in the second experiment (see Table 2): they were larger, they consisted of dynamically linked libraries that provided complex functionality to external applications instead of being a main binary that only performs command-line input-output, and more protections were combined in them. Moreover, the assets protected in the industrial applications are substantially different from the asset (a key) protected in the public challenge, since they consist of an entire functionality (e.g., authentication) of the industrial applications. The higher complexity of the software and the assets protected in the first experiment offers another explanation for the need of more sophisticated and advanced tools and techniques.

Among the attack steps uniquely executed by professional hackers, a common feature seems to be the time and effort devoted to preparation activities. Professional hackers performed several attack preparation activities that were not carried out in the public challenge, such as: *Prepare the environment*, *Preliminary understanding of the software*, *Choose/evaluate alternative tool*, *Port tool to target execution environment*, *Customize execution environment*, *Recreate protection in the small*. This amounts for a lot of work, which is documented in detail in the professional hacker reports and that was necessary to port the attacks. In part such necessity might be due to the complexity of the industrial applications, which cannot be faced directly using off-the-shelf tools, with no preparation or customization. Observing and manipulating dynamically linked Android libraries in action requires a much more specialized environment than doing the same on a simple command-line Linux application. Whereas a more or less standard Linux setup suffices for the latter, a standard Android setup offers very little support to attackers. Another explanation might be that professional hackers are used to think in terms of attack automation. They want their attacks to be repeatable deterministically by anyone, rather than being the result of hardly reproducible, manually executed steps. Hence, they spend some time preparing tools and environment to achieve such level of reproducibility.

There are also a few *tampering* activities that were carried out exclusively by professional hackers, among which: *Tamper with execution environment*, *Replace API functions with reimplementation*, *Out of context execution*. These are all advanced and sophisticated tasks that require deep knowledge and competence. In fact, tampering with the execution environment often involves patching the operating system kernel or developing a new one. Reimplementing API functions also refers typically to operating system level functions that provide basic services and that the attacker wants to replace. Being able to execute a protection mechanism in isolation requires the capability to isolate some functions, mock (some of) the library calls involved and develop a driver that can run the extracted code in isolation. All these activities may be out of reach for non-professional hackers and may pay off only when complex industrial applications are being attacked.

Regarding the software elements that are uniquely used by the professional hackers, it suffices to observe that they mostly correspond to the activities they uniquely reported. Most analysis or tampering techniques target specific kinds of software structures or components. So those structures and components occur in a report when the result or trigger of a discussed attack step is mentioned.

Table 5 (bottom) shows the concepts uniquely used to annotate the public challenge report. A new top level hierarchy emerged, rooted at concept *Difficulty* and including among others *Lack of knowledge* and *Lack of portability*. This concept indicates a problem encountered by hackers during their work (e.g., due to their limited knowledge about a given platform), rather than an *Obstacle* placed there by the defenders to protect the applications or known to be part of the environment, which hence provides some implicit form of protection. The higher relevance of difficulties vs. obstacles in the Public Challenge indicates a non-professional, occasional involvement of public the challenge hacker in attack tasks (confirmed in the interview). Indeed, the winner of the public challenges seems to have quite a different profile than professional hackers.

On the other hand, professional hackers work in team, where the lack of knowledge in a specific field may be compensated by some other members. Moreover, professional hackers have been explicitly selected by the industrial partners because of their specific expertise in attacking their applications. A lack of knowledge or lack of portability, which highlights the impossibility to execute the target application on their systems, were inadmissible.

Some obstacles and weaknesses are specific of the challenges designed for the Public Challenge experiment, such as the *Protection*s *Virtualization* and *Checksum*, and the *Weakness*es *Debug/superfluous features not removed* and *Weak crypto*. That's why they appear only in the Public Challenge annotations.

Similarly, some software elements were uniquely mentioned by the Public Challenge hacker, such as the `main()` function and stderr. This is of course due to the nature of the applications he attacked: simple program binaries that feature a main function and standard command-line input and output, versus libraries as attacked by the professional hackers.

Some attack steps performed exclusively by the Public Challenge hacker seem to indicate an attempt to minimize the attack effort by performing several attack steps manually, rather than trying to automate them (concept *Manually assist the disassembler*), by using functionalities immediately available from tools (concept *Write tool supported script*) and by trying to bypass rather than undo the protections (concept *Bypass protection*). This confirms a different attitude of Public Challenge hackers vs. professional hackers with respect to preparation and automation of the attack: the former try to reuse what's available in tools and compensate for the missing functionalities with manual activities, while the latter spend substantial time preparing the environment and tools for the attack and automating the attack steps.Application size is also a decisive factor that certainly impacted the attack effort minimization choices by the Public Challenge hacker and the professional hackers. The successfully attacked Public Challenge applications were much smaller in size and provided much less functionality than the industrial use cases. This matters because manual and automated attacks scale differently: automation requires a large initial effort to develop tools and scripts. When attacking small applications, in which the occurrences of protection code fragments are by definition limited, the initial automation effort is likely not worthwhile. For small applications, the lack of scalability of manual tampering and information gathering methods is simply not problematic and does not warrant an initial investment in automation.

External factors also affect this different behaviour. Indeed, in the Public Challenge the reward is granted to the first hacker that succeeds in breaking a challenge, therefore, learning and generalizing from the attack tasks and improving their attack arsenal is not their major goal. On the other hand, professional hackers were asked to describe with proper level of details the protection techniques identified and their weaknesses. Moreover, they are usually required (and paid for) writing reports, thus they collect enough information for a complete document. There seems to be also some learning going on for the Public Challenge hacker when moving from one challenge to the next one, as apparent from the following Public Challenge exclusive concepts: *Recognize similarity with already analysed protected application*, *Reuse attack strategy that worked in the past*.

3.3 Inferred Models

To help readability with a not too heavy presentation and to focus on the most interesting findings, we decided not to present and comment all the temporal, causal, conditional and instrumental relations that have been inferred from the hacker reports and their annotations. Since some temporal relations have already been commented during the presentation of the taxonomy of concepts, we do not include this kind of relations. For what concerns the other three kinds of relations emerged during the discussion, we have grouped them by the kind of hacker activity they represent. Hence, they are presented as part of four models: (1) a model of how hackers understand the software and identify sensitive assets (shown in Fig. 6); (2) a model of how they make or confirm a hypothesis, to build their attack strategy (Fig. 7); (3) a model of how they choose, customize and create new tools (Fig. 8); (4) a model of how they defeat protections by bypassing them, by building workarounds, by overcoming protections with other means, and by undoing them (Fig. 9).

The models produced after the experiment with professional hackers have been compared with the new concepts and annotations created in the Public Challenge experiment. It was not necessary to introduce any major change due to the new concepts and annotations, which indicates the general validity and applicability of the original models. A few minor extensions were necessary to accommodate some new concepts introduced after the Public Challenge and relevant for the inferred models. They are commented below at the end of each model description.

3.3.1 How Hackers Understand Protected Software

Let us consider the first model, shown in Fig. 6. Hackers carry out understanding activities with the goal of identifying the sensitive assets in the software that are the target of their attacks. Ultimately, identification of such sensitive assets allows hackers to narrow down the scope of the attack to a small code portion, where their efforts can be focused in the next attack phase (see the *"cause"* relation in Fig. 6). In this process, (static / dynamic) program analysis and reverse engineering play a dominant role. They are used to understand the software, identify sensitive assets, and to limit the scope of the attack (see *"used to"* relation in Fig. 6). For instance, dynamic analysis of IO system calls is used to limit the scope of the attack ([L:D:24] "prune search space for interesting code by studying IO behavior, in

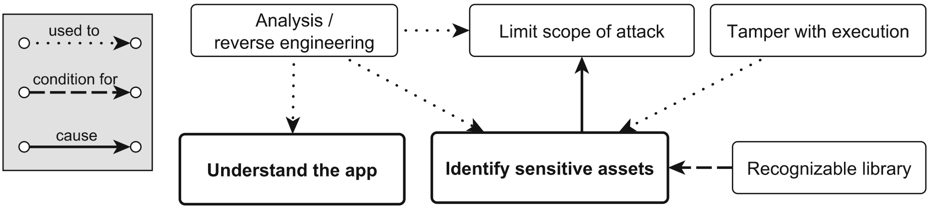


Fig. 6 Model of hacker activities related to understanding the software and identifying sensitive assets

this case system calls"), because some IO operations are performed in the proximity of the protected assets. String analysis is used for the same purpose ([L:D:26] "prune search space for interesting code by studying static symbolic data, in this case string references in the code"), because some specific constant strings are referenced in the proximity of sensitive assets. Tampering with the execution is also a way to identify sensitive assets ([O:E:5] "static analysis + dynamic code injection to get the crypto key"). When libraries with well known functionalities are recognized, hackers get important clues on their use for asset protection (*"condition for"* relation in Fig. 6, based on annotations such as [O:E:6] "static analysis: native lib is using java library for persistence giving clues on data stored to attacker").

The model in Fig. 6 was applied successfully to the public challenge annotations, without any need for extensions. Based on this model, we expect the hackers' task to become harder to carry out when program analysis and reverse engineering are inhibited and when tampering of the program execution is not allowed. In fact, these are the core activities executed to identify sensitive assets and limit the attack scope. Hiding the libraries that are involved in the protection of the assets, not just the protection itself, seems also important to stop / delay hackers.

3.3.2 How Hackers Build Attack Strategies

Figure 7 shows a model of how hackers come to the formulation and validation of hypotheses about protections, and how this eventually leads to the construction of their attack strategy. Hypothesis making requires (see *"cause"* relations in Fig. 7) running (static / dynamic) program analyses and interpreting the results by applying background knowledge on how software protection and obfuscation typically work (e.g., [O:E:4] "static analysis to detect anti-debugging protections"). Identifying protections or libraries involved in protections is also an important prerequisite to be able to formulate hypotheses. When an attack attempt fails (see *"condition for"* relation on the left in Fig. 7), the reasons for the failure often provide useful clues for hypothesis making (sentence *"As the original process is already being ptraced, this prevents a debugger, which typically uses the ptrace system, from attaching"*, annotated as [P:A:50] "Guess: avoid the attachment of another debugger").

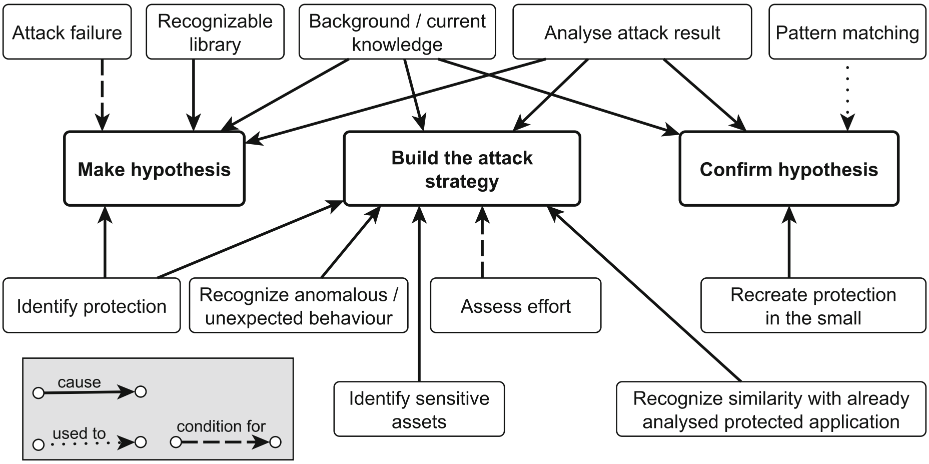

Attack failure
Recognizable library
Background / current knowledge
Analyse attack result
Pattern matching
Make hypothesis
Build the attack strategy
Confirm hypothesis
Identify protection
Recognize anomalous / unexpected behaviour
Assess effort
Recreate protection in the small
Identify sensitive assets
Recognize similarity with already analysed protected application
cause
used to
condition for

Fig. 7 Model of hacker activities related to making / confirming hypotheses and building the attack strategy

To confirm the previously formulated hypotheses, further analyses are run and interpreted based on background knowledge (see *"cause"* relations connected to *Confirm hypothesis*). Pattern matching is also useful to confirm hypotheses ([P:F:26] "Repeated execution patterns are identified and matched against repeated computations that are expected to be carried out by the relevant code"; [P:D:25] "mapping of observed (statistical) patterns to a priori knowledge about assumed functionality"). Another activity that contributes to the confirmation of previously formulated hypothesis is the creation of a small program that replicates the conjectured protection ([P:F:47] "Understanding is carried out on a simpler application having similar (anti-debugging) protection").

Once hypotheses about the protections are formulated and validated, an attack strategy can be defined. This requires all the information gathered before, including the results of the analyses, background knowledge, identified assets and identified protections (see *"cause"* relations connected to *Build the attack strategy*). Another important input for the definition of the (revised) attack strategy is the observation of anomalous or unexpected behaviours (sentence *"[omissis] It seems that the coredump didn't contain all of the process' memory [omissis]"*, annotated as [P:C:31] "Anomaly detected causing doubt in the tool's abilities: change attack strategy"). In fact, unexpected crashes or missing data might point to previously unknown protections that are triggered by the hackers' attempts or to tool limitations. In turn, this leads to the definition of alternative attack paths. Background knowledge plays a major role in strategy building, in particular knowledge about weaknesses (e.g., *Debug/superfluous features not removed* and *Weak crypto*) when these are recognized in the software ("I did the simplest thing possible, the brute-forcing, and the way crypto worked (processing 16 byte blocks independently of each other), meant it was time-inexpensive to do it"). Similarity with previously attacked applications was also found to be an important factor in strategy building in the public challenge experiment (see *"cause"* relation between *Recognize similarity with already analysed protected application* and *Build the attack strategy*).

An important condition that determines the feasibility of an attack strategy is the amount of effort required to implement it (see *"condition for"* relation connected to *Build the attack strategy*). Hence, effort assessment is one of the key abilities of hackers, who have to continuously estimate the effort needed to implement an attack, contrasting it with the expected chances of success ([P:D:51] "assessment of effort needed to extend existing tool to make it provide a workaround around a protection, i.e., defeat the protection that prevents an attack step, in this case based on the concepts of the protection"). Even if potentially effective, attack strategies that are deemed as extremely expensive (e.g., manual reverse engineering and tampering of the code binary) are often discarded to favour approaches that are regarded as more cost-effective.

The validity of the model in Fig. 7 was confirmed by the public challenge experiment, which has provided additional evidence for most of the relations in the model. Only a minor extension was necessary to accommodate a new concept that emerged from the public challenge annotations: the relation between *Recognize similarity with already analysed protected application* and *Build the attack strategy*.

Based on the model shown in Fig. 7, we can notice that hypothesis making and attack strategy construction are inhibited by the same factors that inhibit software understanding and sensitive asset identification. In addition, a further factor comes into play: the estimated *effort* to implement an attack. Hence, even protections that can be eventually broken play potentially a key role in preventing attacks, if they contribute to increase the effort required for attacking the target sensitive assets.

3.3.3 How Hackers Choose and Customize Tools

Hackers extensively use existing tools for their attacks. An important core set of the hackers' competences is deep knowledge of tools: when and how to use them; how to customize them. Figure 8 shows how hackers evaluate, choose, configure, customize, extend and create new tools (see also sub-concepts of *Customize/extend tool*, including *Write tool supported script*). The starting point is usually the result of some analysis and/or the observation of some specific obstacle, which leads to the identification of candidate tools (see *"cause"* relations in Fig. 8). Then, a key factor that determines both tool selection and customization is the execution environment and platform. Other important factors are known limitations of existing tools, which might be inapplicable to a specific platform or application ([P:A:23] "[omissis] Attack step: dynamic analysis with another tool on the identified parts to overcome the limitation of Valgrind"), as well as observed failures of previously attempted dynamic analysis ([P:C:38] "Experiment with tool options to try to circumvent failures of the tool"), which may suggest alternative approaches and tools (see *"condition for"* relations on the left in Fig. 8).

Once tools are selected and customized, they are used to find patterns, by running further analyses on the protected code, or they are used directly to undo protections and mount the attacks (see *"used to"* relations in the middle of Fig. 8). When existing tools are insufficient for the hackers' purposes, new tools might be constructed from scratch. This is potentially an expensive activity, so it is carried out only if existing tools cannot be adapted for the purpose in any way and if alternative tools or attack strategies are not possible. One case where such tool construction from scratch tends to be cost-effective is when hackers want to execute a part of the software out of context, to better understand its protections (see *"used to"* relation connected to *Out of context execution*). In fact, this usually amounts to writing scaffolding code fragments that execute parts of the application or library under attack in an artificial, hacker-controlled, context ([L:E:17] "write custom code to load-run native library").

The model in Fig. 8 was fully applicable to the public challenge annotations, with no need for any extensions. The public challenge experiment provided substantial further support to the general validity of this model. The model shows that tools play a dominant role in the implementation of attacks. Hence, software protections should be designed and realized based on an amount of knowledge of tools and of their potential that should be as deep and sophisticated as the hackers' one. Preventing out of context execution is another important line of defence against existing and new tools.

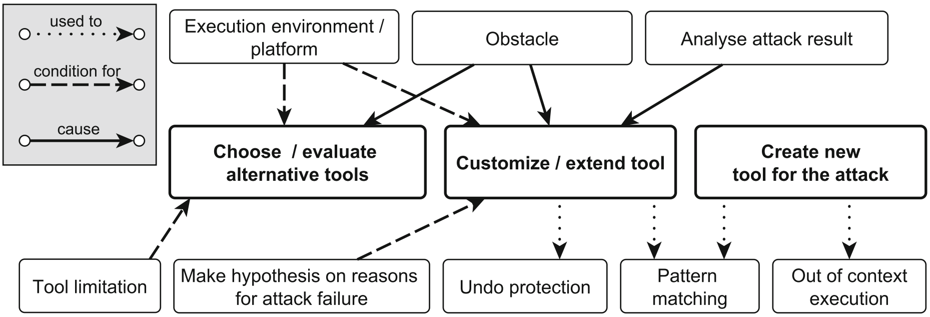


Fig. 8 Model of hacker activities related to choosing, customizing, and creating new tools

3.3.4 How Hackers Defeat Protections

The actual execution of an attack against a protection aims at defeating it, by bypassing it, building a workaround, undoing the protection completely, or overcoming it in some other way. Figure 9 shows a model of such activities.

Undoing a protection is usually regarded as quite difficult and expensive. In some cases, hackers opt to overcome a protection by tampering with the code or the execution (see incoming relations of *Overcome protection* in Fig. 9). This means that instead of reversing the effect of a protection (e.g., deobfuscating the code), they gather enough information to be able to manipulate the code and the execution so as to achieve the desired effect, without having actually removed the protection. Gathering the information and performing the manipulations with the protections still present typically requires a considerable effort in analysis, and in building external tools, scripts, or tool extensions. Overcoming a protection eventually relies on the possibility to alter the normal flow of execution, this is the reason for a causal relation between *Tamper with execution* and *Overcome protection*.

In some instances, altering the execution flow with external tools is not enough, not possible, or requires too much effort. In such cases, hackers may write custom workaround code (*Build workaround*) that is integrated with or replaces the existing code, with the purpose of preserving the correct functioning of the software, while at the same time making the protections ineffective.

Sometimes hackers run program analyses to obtain information that is useful for manually undoing protections. For instance, dynamic analysis and symbolic execution can be used to understand if a predicate is (likely to be) an opaque one, such that one of the two branches of the condition containing the predicate can be assumed to be dead code that was inserted just to obfuscate the program ([L:F:2] “Undo protection (opaque predicates) by means of dynamic analysis and symbolic execution”). The analyses needed to undo protections may be quite sophisticated, hence requiring non trivial tool customization (see incoming relations of *Undo protection* in Fig. 9).

To overcome a previously identified protection, hackers alter the execution. For instance, if they have identified some library calls used to implement a protection, they may try to intercept such calls and replace their parameters on the fly; they may skip the body of the called functions and return some forged values; or, they may redirect the calls to other

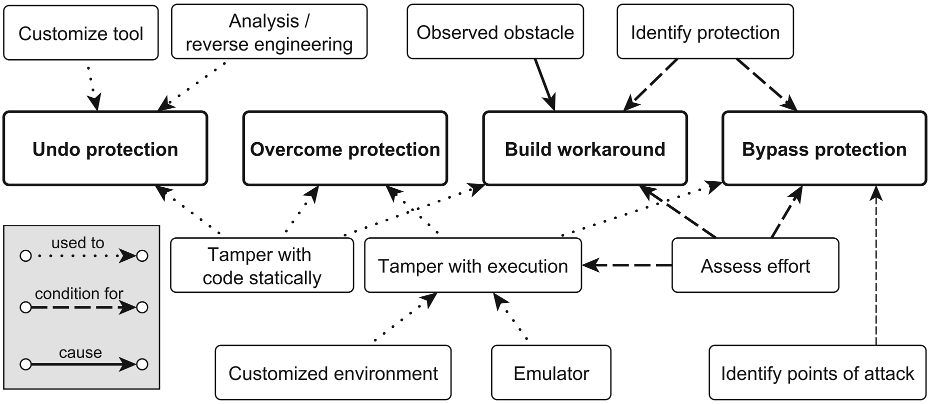


Fig. 9 Model of hacker activities related to defeating protections by undoing, overcoming, working around, or bypassing them

functions ([O:F:17] “Tamper with system calls (ptrace) that implement the anti-debugging protection by means of an emulator”; see causal relation to *Overcome protection* in Fig. 9). To achieve the desired effect, this might require also altering the code (see *“used to”* relation to *Overcome protection*; [L:F:7] “Tamper with protection (anti-debugging), by patching the code [omissis]”).

Tampering with the execution can be more or less expensive, depending on the intended manipulation of the execution ([P:C:48] “Investigate API usage of the protection to see how much effort it is to emulate it”). For this reason, a key decision support activity is *Assess effort* (see *“condition for”* relation at the top in Fig. 9). In practice, implementing execution tampering requires non-trivial skills on tools such as emulators, instrumentation tools, and debuggers, and on the customization of the execution environment (see *“used to”* relations at the top). Hackers may even resort to a custom execution kernel ([O:F:18] “Tamper with system calls (ptrace) that implement the anti-debugging protection by means of a custom kernel”).

Whenever protections cannot be easily undone or overcome (with the help of existing tools), hackers build workarounds. Hence, the trigger for this activity is an observed obstacle that cannot be defeated by simpler means (see *“cause”* relation at the top in Fig. 9). Since building workarounds is typically an expensive activity, effort estimation is routinely conducted before starting this attack step (see *“condition for”* relations at the top; [P:A:51] “Identification of a potential trick to avoid the protection technique. Estimation of the consequences (scripts and time wasted on this)”). Moreover, identifying the specific protection (component) to defeat is a prerequisite for the construction of the proper workaround (see *“condition for”* relation at the top in Fig. 9). For instance, hackers may intercept decrypted code before it is executed rather than trying to decrypt it ([O:G:3] “[omissis] Bypass encryption; weakness: decryption before execution”). In the Public Challenge experiment, we found cases where no workaround was actually necessary and the protection could be bypassed directly from the debugger, by simply tampering with the data at run time. This justified the introduction of a new concept, *Bypass protection*, which is a viable attack technique when protections and points of attack have been identified, and the effort to actually bypass the protection is deemed acceptable (see incoming relations of concept *Bypass protection* in Fig. 9).

Based on the model of execution tampering to undo, overcome, work around, or bypass protections shown in Fig. 9, we can again notice that effort assessment is a key activity that is carried out continuously. Moreover, such continuous effort estimation leads hackers to prioritize their attack attempts. If undoing a protection is regarded as too difficult and too effort intensive, hackers may switch to limited code tampering or dynamic manipulation of the execution, so as to overcome, work around, or bypass the protection without reverting it completely. When all of those methods fail or are deemed too much demanding in terms of effort, attackers can opt for circumventing a protection by deploying alternative attack approaches that are not hampered by the particular protection. Making such decisions is a standard activity when attackers choose the path of least resistance.

4 Discussion

4.1 Research Agenda

Based on the observed attack steps and strategies, we have identified the following research directions for the development of novel or improved code protections.

Protections Should Inhibit Program Analysis and Reverse Engineering (see *"used to"* relation outgoing from *Analysis / reverse engineering* in Fig. 6). While several of the existing protections are designed to inhibit program analysis (e.g., control flow flattening; opaque predicates) and (manual) reverse engineering (e.g., variable renaming), in our study we have noticed that hackers employ advanced program analysis techniques, like dependency analysis, symbolic execution, and constraint solvers. These techniques are indeed very powerful, but they come with known limitations. For instance, dependency analysis is difficult when pointers or reflection are extensively used; symbolic execution is difficult when loops and black box functions are used; constraint solvers may fail if non-linear or black-box constraints are present in expressions. Protection developers may exploit such limitations to artificially inject constructs that are difficult to analyse into the program. Since manual intervention might be needed to help tools deal with such artificially injected constructs, it would be interesting to perform an empirical study to test the effectiveness of such solutions. Such a study may compare, both qualitatively and quantitatively, a control group and a treatment group, which attack software respectively without/with artificially injected constructs. The two groups would be allowed to use the same program analysis tools. The qualitative comparison could be focused on the difference in attack strategies and manual comprehension steps between the two groups.

Protections Should Prevent Manipulation of Both the Execution Flow and the Run-Time Program State (see relations outgoing from *Tamper with execution* in Figs. 6 and 9). Intercepting the execution and replacing the invoked functions or altering the program state is a key step in most successful attacks. Protections that inhibit debuggers (e.g., anti-debugging techniques) or that check the integrity of the execution (e.g., remote attestation) are hence expected to be particularly important and effective. Another approach to prevent execution tampering is the use of a secure virtual machine for the execution of critical code sections. Our study provides empirical evidence on the importance of pushing these research directions even further. Human studies could be designed to determine the strategies adopted by attackers to defeat each of the above mentioned protections. Such empirical studies would be also useful to assess quantitatively the relative strength of the alternative protections.

Protections Should Adopt Integrity Checking Techniques with Improved Effectiveness (see the "condition for" Relation from Identify Point of Attack and Bypass Protection) Both the applications used for the Public Challenge and the ones for the experiment with Professional Hackers include code guards to verify the integrity of the executed code. Code guards verify checksum values at run time, and in some cases, if values are correct the application execution continues, otherwise, the application could crash or gracefully degrade its performance. Attackers were able to obtain the correct checksum values by setting breakpoints on valid applications and injecting them with a debugger when requested by the tampered applications, as precisely described in the Public Challenge report. These lead to the observation that attackers can easily bypass these integrity protections. Therefore, research should focus on improving the effectiveness of integrity checking techniques. Research is progressing in this field with remote guards and software attestation techniques (Armknecht et al. 2013), which allow the use of nonces to add "freshness" at every guard check. Even in this case, using static values (like binaries) is the main limitation, as variants of the Van Oorschot attack are possible (van Oorschot et al. 2005). It is therefore mandatory to build new techniques that use dynamic/run-time information for attesting the application integrity.

Diversification and Hiding of the Fingerprints of Protections (see the "cause" Relation from Recognize Similarity with Already Analysed Protected Application to Build the Attack Strategy in Fig. 7) Protections, especially when automatically applied by a tool-chain, use the same process and methods to protect the application, but they usually randomize it, every time with a different seed. On both the Professional Hackers and the Public Challenge Winner reports, we noticed that experienced attackers (this might be generalized to smart occasional attackers) easily recognise when the same protections are applied to different applications, despite the randomization. Therefore, they can immediately *Build attack strategies* that aim at reusing methods and tools already used (and that worked) in the past. Reapplying the same strategies usually does not require a deep understanding of the code and functionality of the protected application, and results in a very fast successful attacks. Even if we already noticed in our study that hackers try to *Recognise libraries* and *Anomalous/unexpected behaviour* and perform *Pattern matching*, further human experiments can help understanding which are the characteristics of the fingerprints that can be noticed by estimating the learning effect of undoing, bypassing, or removing the same protection when applied on different code (from different applications). Based on these empirical results, it would be possible to design methods to diversify and hide the fingerprints of protections thus forcing hackers to resort to other strategies that require more time and effort.

Libraries Involved in Code Protections Should be Hidden (see Relations Outgoing from *Recognizable library* in Figs. 6 and 7) Libraries represent a side channel for attacks that is often overlooked by protection developers. Our study shows that protecting the code of the main application is not enough and that the libraries used by the application and referenced in its code may leak information useful to hackers and may offer them viable attack points. Techniques to prevent attacks to libraries and to obfuscate the use of libraries or the libraries themselves deserve more attention from protection developers. Moreover, vulnerability indicators and metrics could be defined to determine the occurrence of libraries, system calls and external calls, which can be regarded as potential points of attack.

Protections Should be Selected and Combined by Estimated Attack Effort (see Relations Outgoing from *Assess effort* in Figs. 7 and 9) The (theoretical) strength of a protection is of course important when deciding to apply it, but according to our study the perceived effort to defeat a protection is even more important (and indeed it may differ from the theoretical strength). The perceived strength is defined as subjective evaluation of the impact a protection may have on hackers decision on how to tamper with the application (*Build the attack strategy*). This means that even theoretically weak protections (e.g., variable renaming) should be used as they increase the attack effort and may discourage some faster attack strategies. Hence we aim at estimating the perceived attack effort through novel attack effort prediction metrics. Moreover, synergies among protections should be investigated. Synergies may increase the effort necessary to defeat protection more than the sum of the attack effort required by each protection alone (e.g., apply code guards to render the code modified for out of context execution purposes unusable, together with obfuscation to render modifications even more complex). To actually prioritize the protections to apply, more effective metrics that estimate the actual potency of protections would be needed, either when applied in isolation or in combination with other protections. Moreover, it would be interesting to empirically assess the correlation between such metrics and the actual delays introduced by protections. The correlation between perceived strength and actual delay would also be worth extensive empirical investigation.

Effectiveness of Protections Should be Tested Against Features Available in Existing Tools or by Customizing Existing Tools (see *Choose Tool / Evaluate Alternative Tools* and *Customize / Extend / Configure Tool* in Fig. 8) While this practice might sound quite obvious, in our experience it is overlooked by protection developers, who usually assess the strengths of protections either theoretically or through metrics. Empirical evaluation based on deep knowledge and customization of existing tools may provide useful insights for the improvement of the proposed techniques. This consideration also applies to all the tools able to *Undo protections*, like deobfuscators. Indeed, undoing a protection may be very time consuming, but only if there are no tools able to automatically perform this attack step.

Out of Context Execution of Protected Code Should be Prevented (see *Out of Context Execution* in Fig. 8) This attack strategy is not much known and investigated, but in our study it appeared to play a quite important role. Protection developers should design techniques to make the protected code tangled with the rest of the software, so as to make out of context execution difficult to achieve. A human study could be conducted to measure the difficulty of out of context execution when the protected code is made arbitrarily tangled with the rest of the software in comparison with the initial, untangled code.

Protections Should be Difficult to Defeat Without Rewriting Part of the Code as a Workaround to them (see *Build Workaround* in Fig. 9) While the perfect protection for a software asset may not exist (Barak et al. 2001), practical protections should be designed such that the only way to defeat them is writing substantial code (e.g., a new library, a new kernel, a replacement function, etc.). In fact, this increases the attack effort and deters or defers the attack. What workarounds hackers write in practice and how they elaborate them is yet another research topic on which little is known and that would deserve further investigation.

Empirical Validation of Protections Should Involve Highly Trained Subjects Playing the Role of Hackers (see the Comparison Between Professional Hackers and Public Challenge Winner Reported in Section 3.2) Differently from the amateur hacker, professional hackers aim for automated and reproducible attacks, which require substantial preparation of environment and tools, use of sophisticated and advanced analyses / techniques, deep knowledge and competence. In order for an empirical validation of protections to produce results that generalize to professional hackers, the involved subjects should be trained to a comparable level of competence. This requires training on advanced analysis and techniques, such as symbolic execution, SMT solving, cryptanalysis, data flow analysis, differential data analysis, correlation analysis. It also requires training on the tools most widely used by professional hackers, in particular debuggers (see *Tool* hierarchy in Fig. 4), and it requires strong capabilities to choose/customize/port existing and to develop new tools in preparation for the attack. Automation of the attack steps is another key difference between professional and occasional hackers, which deserve attention during training for an empirical study. Advanced tampering activities, such as tampering with the execution environment, replacing API functions with reimplementations and executing the protection out of context, may require dedicated training, since they seem not to be part of the toolkit commonly available to the occasional hacker.

Protections Should Force Tools to Produce Incorrect Code Representations and Structures When we discussed our results with professional (malware) reverse engineers, it became clear that protections to mitigate or to complicate and delay reverse engineering should not only aim for making the program representations, elements and structures (see Fig. 5) harder to obtain (e.g., by blocking the use of tracing tools or debuggers), to identify (e.g., by injecting many fake XOR operations), and to interpret (e.g., by means of control flow flattening). When the protections only focus on those attack processes, such that, e.g., the control flow graph obtained with a disassembler tool like IDA Pro is incomplete, attackers will typically either manually complete the control flow graph (via the tool's interactive functionality) or they will write scripts to extend the tool (via its built-in scripting support) such that it produces a more complete control flow graph in later re-runs. The attackers are hence moving forward on their attack path, implementing an attack strategy and gradually obtaining more complete and more accurate information. In other words, they are unlikely to start unsuccessful attack paths based on incorrect assumptions. In fact Piorkowski et al. showed that developers tend to stubbornly follow even wrong cues (Piorkowski et al. 2013), moreover Edmundson et al. (2013) showed that even experienced developers tend to overlook important information.

Protections become stronger if they also succeed in letting the tools produce incorrect representations and structures, e.g., by including non-existing edges in a control flow graph or by grouping code fragments incorrectly into functions. Attackers then determine their strategy and next attack steps based on incorrect information, because their default modus operandi is to initially accept the information obtained from the tools as is, and to build on it as is. When that information is incorrect, they will hence more likely waste time and effort on unsuccessful attack paths before backtracking, and before questioning the correctness of the information, identifying the issues in it, and eventually correcting them.

4.2 Threats to Validity

External validity (concerning the generalization of the findings): The purpose of our qualitative studies was to infer models of the hackers' activities starting from the hacker reports. Being the result of an inference process grounded on concrete observations, our models may not have general validity. Further empirical validation is needed to extend the scope of their validity beyond the context of the reported studies. However, during conceptualization we aimed explicitly at abstracting away the details, so as to distill the general traits of the ongoing activities. Moreover, in order to obtain models that are applicable in an industrial context, the first experiment was conducted in a realistic setting, involving professional hackers who are used to perform similar attack tasks as part of their daily working routine.

Despite the substantially different setting of the Public Challenge, a proportion of concepts (41%) could be reused to annotate the new report. This indicates a reasonable level of generality of the taxonomy and provides a first, initial positive assessment of its external validity. Fourteen percent new concepts had to be added to the taxonomy. This was expected, because, according to our initial design, the context and hacker profiles of the second study are remarkably different from those of the first study (amateur vs. professional hackers, different combination of protections). The fact that the same person won all five public challenges has certainly limited the variability and expressiveness of the narrative we have experienced in the public challenge report. However, we cannot give objective indications on how the proportion of reused concepts would have changed with more winners neither can we say if changes in the narrative would have resulted in an increased number of new concepts. For instance, if the winner had a more professional profile, we could have experienced an increase in the reused concepts, while reports from more amateur hackers could have led to a slightly higher number of new concepts. On the other hand, having a single winner reduced the issues related to the comparison of reports from multiple persons thus reducing the threats to the external validity.

Hence, we cannot claim to have reached theoretical saturation. New experiments are needed to further expand the concept ontology. Still, reuse of a substantial proportion of concepts in the second experiment, despite the differences, is encouraging.

Data collected in our experiments only refers to the client-side component of software systems, because in our settings all the server-side component are considered out of the attack scope.

Construct validity (concerning the data collection and analysis procedures): We adopted widely used practices from grounded theory to limit the threats to the construct validity of the study. To be sure that reports contain all the needed information, we asked professional and public challenge hackers to cover a set of topics while filling their reports, including obstacles, activities, tools and strategies, and to answer specific questions.

Internal validity (concerning the subjective factors that might have affected the results): In order to avoid bias and subjectivity, in the first experiment coding was open (no fixed codes) and it was performed by the seven coders independently and autonomously. Moreover, precise instructions have been provided to guide the coding procedure. To complete concept identification and model inference, two joint meetings have been organized. All the interpretations were subject to discussion, until consensus was reached. Traceability links between report annotations and abstractions have been maintained. This was effective not only to document decisions, but also in case of model revision, to base changes on evidences from the reports. In the second experiment coding was closed, except for the new concepts that were added to the taxonomy, for which it was open (as in the first experiment). Similarly to the first experiment, precise coding instructions have been provided to the four teams of annotators and two consensus meetings have been carried out to converge to an agreed annotation of the Public Challenge report. While some subjectivity is necessarily involved in the process, the above mentioned practices aimed at minimizing and controlling its impact.

5 Related Work

The related literature consists of the empirical studies conducted to produce a model of program comprehension and of the developers' behaviour. Empirical studies on the effectiveness of software protections are also relevant.

5.1 Models of Program Comprehension and of the Developers' Behaviour

The construction and validation of program comprehension models have been the topic of investigation of a huge amount of works since the beginning of the discipline of software engineering. There is agreement, based on experimental observations (von Mayrhauser and Vans 1995), that program comprehension involves both top-down and bottom-up comprehension, and that often the most effective strategy is a combination of the two, known as the integrated model (von Mayrhauser and Vans 1994, 1996a). A less systematic combination of top-down and bottom-up models is called the opportunistic model of program comprehension (von Mayrhauser and Vans 1995).

Programmers resort to a top-down comprehension strategy when they are familiar with the code. They take advantage of their knowledge of the domain (Pennington 1987) to formulate hypotheses (Letovsky 1987) that trigger comprehension activities. Hypotheses (Letovsky 1987) can take the form of *why/how/what* conjectures about the expected implementation. Comprehension activities carried out on the code aim at verifying the hypotheses on which uncertainty is highest. Upon resolution of such uncertainty, programmers build a complete, mental top-down model of the program, consisting of a hierarchy in which the lowest level hypotheses have been validated against the implementation and mapped onto the code. When hypotheses fail to be verified, the top-down mental model is iteratively refined until convergence to a stable model.

The bottom-up strategy is preferred by programmers who are relatively unfamiliar with the code. Programmers may start with the construction of a control flow model of the program behaviour (Pennington 1987), to continue with higher level abstractions, such as the data flow model, the call hierarchy model and the inter-process communication model. Eventually, programmers obtain a high level, functional abstraction of the implementation.

In the integrated model (von Mayrhauser and Vans 1994), programmers work at the abstraction level that is deemed appropriate for the task at hand and switch between top-down and bottom-up models. They recognize clues (aka beacons) in the code that point to higher level abstractions and then they switch to a top-down comprehension of the abstractions inferred from such clues. This leads programmers to formulate new hypotheses to be verified in the code, which in turn trigger a new bottom-up phase, consisting of code search and clue recognition. While the bottom-up phase is usually quite systematic, the top-down phase tends to be opportunistic and goal driven (von Mayrhauser and Vans 1997a, b). The opportunistic strategy has been found to be much more effective and efficient than the systematic strategy when applied to large systems (Littman et al. 1987). However, it has the disadvantage of producing incomplete models and partial understanding, which might affect negatively program modification (von Mayrhauser and Vans 1994, 1996a, b).

Existing program comprehension models have been investigated in specific contexts, such as component based (Andrews et al. 2002) or object oriented (Burkhardt et al. 2002) software development, but to the best of our knowledge ours is the first work considering the comprehension process followed by professional and public challenge hackers during understanding of protected code to be attacked. The work by Sillito et al. (2006) investigates general traits of program comprehension that are common to our observations. However, some activities that we observed are hacker specific and driven by the hackers' goal, which is to break a protection, not to evolve a system, e.g., for adding new features, fixing bugs or refactoring/adapting the code.

5.2 Qualitative Studies of the Developers' Behaviour

The use of qualitative analysis methods in software engineering has gained increasing popularity in recent years and among the various qualitative methods, GT appears to be the most popular one. However, according to a survey (Stol et al. 2016) conducted on 98 papers that have been published in the 9 top ranked journals, GT is largely misused in existing software engineering studies. In fact, key features of GT, such as theoretical sampling, memoing, constant comparison and theoretical saturation, are often ignored. We do not claim to have used GT in our study. Instead, we claim that some practices that we adopted are in common with GT. Other practices could not be applied because of limitations of our experimental settings. The GT survey (Stol et al. 2016) reports some of the topics investigated in the 98 qualitative empirical studies analysed for the survey. These include studies on the software engineering process and on software development teams, especially in the agile context. For instance, the paper by Prechelt et al. (2016) investigates the factors that affect software quality in agile teams without dedicated testers. No mention is made in the survey of any qualitative analysis dealing with the program comprehension process carried out by hackers. Our search for papers on such topic has also produced no result.

In other domains in computer security, such as network penetration cyber attacks, qualitative modeling techniques have been proposed (Katipally et al. 2011; Mallikarjunan et al. 2015; Vectra 2017). Those models and modeling techniques specifically focus on automated intrusion detection, however. The concepts in them are observable symptoms of attack activities in staged network penetrations. By checking whether or not monitored network activities fit within the models of different types of attackers, network administrators and their decision support tools can then distinguish benign activities from ongoing attacks. The concepts and relations in those models are of a completely different nature, and hence not reusable to model program comprehension activities. If program comprehension is certainly needed to prepare such attacks (i.e., to spot a bug), none of the existing works about network penetration report any information.

Our previous experiment (Ceccato et al. 2017) of hackers' code comprehension has been extended in the present work with an additional experiment based on a public challenge. The report of the public challenge winner was subjected to the same kind of qualitative analysis conducted on the professional hacker reports. The additional results have confirmed the validity of the taxonomy and of the comprehension models extracted from the initial experiment, but they also allowed us to extend the taxonomy and the models with new concepts that emerged in the public challenge context, a context substantially different from the industrial one considered in the first experiment. The substantial number of concepts identified with the second experiment suggests that theoretical saturation is not achieved, and that more experiments are needed to enrich the concept ontology.

5.3 Empirical Studies on the Effectiveness of Software Protection

There are two main research approaches for the assessment of obfuscation protection techniques, respectively based on internal software metrics (Collberg et al. 1997; Ceccato et al. 2015; Anckaert et al. 2007; Linn and Debray 2003; Udupa et al. 2005; Capiluppi et al. 2012) and on experiments involving human subjects (Sutherland et al. 2006; Ceccato et al. 2009, 2014, Viticchié et al. 2016).

Assessment by means of experiments with human subjects has been first presented in a work by Sutherland et al. (2006), who found the expertise of attackers to be correlated with the correctness of reverse engineering tasks. They also showed that source code metrics are not good estimators of the delays introduced by protections on attack tasks, if binary code is involved. Ceccato et al. (2009) measured the correctness and effectiveness achieved by subjects while understanding and modifying decompiled obfuscated Java code, in comparison with decompiled clear code. This work has been extended with a larger set of experiments and additional obfuscation techniques in successive works (Ceccato et al. 2014; Viticchié et al. 2016).

While human experiments conducted to measure the effectiveness of protections often draw also some qualitative conclusions on the activities carried out by the involved subjects, their main goal is not to produce a model of the comprehension activities carried out against the protected code. Moreover, the involved subjects are usually students, not professional hackers. Hence, while these studies contributed to increase our knowledge of the effectiveness of various software protection techniques, they did not develop any thorough model of code comprehension during attack tasks.

6 Conclusions and Future Work

The goal of this work was to investigate the way hackers comprehend protected code to be attacked and how they build their attack strategy. We involved professional hackers in the execution of three industrial case studies, each with a distinct attack task. We also published eight public challenges, five of which were broken, all of them by the same hacker. The reports produced by the professional hackers and by the public challenge winner have been subjected to a rigorous qualitative analysis, resulting into a taxonomy of concepts and in four behavioral models. The taxonomy consists of 169 concepts associated with the hacker comprehension, attack activities, analysis tools and identified protections.

The models introduce relations (e.g., temporal, causal, conditional and instrumental relations) among concepts to explain hackers behavior, i.e.:

- how hackers understand the code and identify sensitive assets within it;
- how hackers formulate and confirm hypotheses to build their attack strategy;
- how hackers choose and customize tools; and
- how hackers defeat protections by undoing, overcoming, working around, and bypassing them.

The paper presents the commonalities and the differences emerged between the activities conducted by professional hackers (involved in industrial settings) and by the practitioner (involved in the Public Challenge). A major difference is represented by the level of automation and reproducibility of attacks elaborated by professional hackers, and in their considerable time investment in constructing new tools to achieve this objective. Practitioners, instead, prefer to try many fast manual attacks or employ existing general purpose tools.

The paper includes a discussion of possible research directions in code protection based on the outcome of the empirical investigation. One of them is how to design novel and stronger protection techniques. Since automated analysis tools play a central role in elaborating successful attacks, a promising research direction to investigate is improving protection techniques by addressing known (theoretical or practical) tool limitations, in order to increase the manual effort in attacks.

While we believe the taxonomy and models have a general validity, it can be strengthened only by conducting further empirical studies. The relations in our models that enable or prevent hacker activities will be investigated in depth with controlled experiments, to obtain quantitative evidence on their role and power, which in turn will guide the development of novel code protection techniques.

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.

Glossary

Analyse attack result	Analyse the knowledge obtained from perform an attack step or combination of attack steps
Anti-callback stack checks	Checks performed on entry to internal functions to verify that they are not called from externally
Anti-debugging	Form of software protection that makes it harder for an attacker to attack a debugger to an executing software to attack it
Assess effort	Reason about effort needed to perform an attack step
Asset	Software artefact valuable to attackers, typically the artefacts of the original software on which security requirements are formulated,

	plus potentially the protections deployed on them in so far as those protections become under attack as well
Background knowledge	Relevant knowledge attackers have before starting attack activities on a specific piece of software
Basic block	Sequence of instructions that can only be executed as a whole and in that order
Black-box analysis	Analysis techniques that only consider the input-output behavior of the program or a component thereof
Brute force attack	Try all inputs on a code fragment to obtain the desired input or output or behavior
Build the attack strategy	The making of a decision regarding the next attack steps to be executed
Build workaround	Defeating a protection by developing custom code integrated in, or replacing parts of, the software under attack (including the protection in it under attack) to make a protection ineffective (possibly leaving the original software functionality intact) such that an attack step can be executed.
Bypass protection	Using debugger commands (or scripts) or other lightweight techniques to manipulate the execution of the software without altering its code, thus making the protection ineffective, e.g., by allowing the software execution to progress nominally beyond the point where the protection was supposed to intervene.
Bytecode	Non-native instruction set architecture format
Checksum	Hash computed over some data, in this context typically a code region being checked by a code guard
Choose path of least resistance	Reason about and select the potentially successful sequence of attack steps that will lead to reaching the overall attack goal with the least effort or cost
Choose/evaluate alternative tool	Check whether alternative tool overcomes a limitation of a previously tried tool
Circumvent protection	When a protection prevents reaching a goal with one or more specific attack steps, execute one or more alternative steps that are orthogonal to the deployed protection (i.e., not hampered by the protection) to reach the same goal
Clear data in memory	Asset other than key appearing in unprotected (plaintext) form in binary or during execution
Clear key	Cryptographic key appearing in unprotected form in binary or during execution
Client-server code splitting	Protection whereby part of the sensitive computations is extracted from a client program and executed on a secure server instead, where it is not accessible to attackers
Clues available in plain text	Presence of strings in binary that present clues about protections, assets, components, ...
Code guard	Specific form of tamper detection that computes a checksum on code regions and checks their values by comparing them to precomputed ones
Code mobility	Online protection in which static code (and data) fragments are removed from an executable file (to prevent static attacks) and instead get downloaded from a secure server on demand during the execution of the software
Code representation and structure	Static or dynamic representations of software (fragments) and structural forms or elements in it
Condition	(combination of) values that inputs, variables, registers, memory locations, ... need to have in order to let control be transferred into a certain direction during execution of the software
Confirm hypothesis	Confirm that a previously hypothesized feature is correct based on the observation of an attack step's results
Constant	Static data with relevant or recognizable, non-mutable values
Control flow graph	Static representation of potential control flow in a program

Control flow graph reconstruction	Determining and modelling the potential flow of control in a program or part thereof
Convert code to standard format	The act of converting (byte)code in a custom (diversified) format to a format known by the attacker
Core dump	Snapshot image of the software's memory space during its execution, containing code and data segments such as stack, heap, and code sections from binaries
Correlation analysis	Statistical analysis where correlations between operations or data occurrences are determined or analysed
Correlation between observed values	The presence of statistical correlation in a set of data values observed in a program, memory, or a trace
Create new tool for attack	Create new standalone piece of software to aid in attack (e.g., a main binary that invokes functionality in library under attack in a specific order and that feeds it specific data)
Crypto analysis	Use of cryptanalysis techniques
Customize execution environment	Adapt software or hardware in execution environment in ways supported by their developers such that it supports specific attack tasks
Customize/extend tool	Adapt tool in ways supported by tool developers and tool itself (e.g., availability source code) or exploit its built-in extensibility to let it perform specific tasks
Data and program state	Static or dynamic non-code aspects of a binary or running process
Data flow analysis	Analysis techniques that determine how computations and computed data depend on other computations and (computed or input) data
Deobfuscate the code	Use manual or automated tools to revert an obfuscation, i.e., to reduce its apparent complexity to that of the original , non-obfuscated code
Debug/superfluous features not removed	Functionality present for software development purposes (e.g., debugging aids) that was not removed before distributing the binaries and that can be leveraged by attackers
Debugger	Tool used to test and debug software by offering support to inspect and manipulate the status of running software
Debugging	Using debugger functionality to observe, control, and manipulate a program's execution
Decompile the code	Obtain source code equivalent of machine code
Decompiled code	Representation of binary software code at the abstraction level of source code
Decompiler	Software that translates assembly language into equivalent source code
Decrypt code before executing it	(large) code fragments only available in encrypted form become available in decrypted form at run time
Defeat protection	Successfully undo or overcome or bypass a protection, or build a workaround for it, such that an attack step that the protection was supposed to mitigate can be executed successfully.
Dependency analysis	Analysis techniques that determine which computations and computed data depend on which other computations and (computed or input) data
Difference between observed values	The fact that two values in a trace, binary, or memory have different values
Differential data analysis	Statistical analysis where not the original observed operations or data are considered but differences between multiple occurrences
Difficulty	Problem encountered during an attack task that is caused by a feature, artefact or limitation of the attacker's toolbox, i.e., of the specific software or hardware attack aids being used or considered during a concrete attack (that are not normally used during benign use of the software under attack).
Diffing	Identifying the differences between two programs or parts thereof
Disable anti-debugging	Tampering with code to skip execution of anti debugging protection actions

Disassemble the code	Use a tool to convert binary encoding of software into human readable machine code
Disassembled code	Representation of binary software code at the abstraction level of assembly code
Disassembler	Software that translates machine language into assembly language (and determines the structure thereof, e.g., in the form of control flow graphs)
Diversified cryptographic libraries	Libraries with non-standard implementations of standard cryptographic primitives
Dynamic analysis	Analysis techniques based on observations made during program execution
Emulator	Hardware or software that enables one computer system to behave like another system
Evaluate and select alternative step/ revise attack strategy	Reason about effort, success probability, usefulness, ... of possible next attack steps, building on results of previous attack steps, and select next steps / revise decisions and selections made earlier regarding next attack steps to execute
Execution environment	Operating system, platform, network settings, etc. in which the code has to be executed
File	A resource for storing information, typically on a storage device
File name	Symbolic identifier of a file
File format analysis	Black-box analyses that consider the formats of input and output files
Function / routine	Software components making up programs in most programming languages
Function argument	Data (and value thereof) on which a function is invoked
Function call	The operation of invoking a callee function within a caller function
Global function pointer table	Standard data structures in binaries that contain addresses of functions
Identify API calls	Act of identifying locations in the code or trace, and their nature, where interaction with public interfaces of external components take place
Identify assets by naming scheme	Use naming conventions or structure in available symbol information
Identify assets by static meta info	Use standard information available in binaries (e.g., exported symbols) to identify components that embed assets
Identify output generation	Identification of code around points where output is generated as starting points of attacks
Identify points of attack	Identify regions in the program or trace where assets or protections are available/observable/active/... and hence attackable
Initialization function	Function invoked by loader upon loading of a program or library
In-memory data structure	Data structure found in the memory space of executing software
Knowledge on execution environment framework	Relevant knowledge attackers have about the execution environment or framework in which they will execute attack steps or in which the software normally executes
Lack of knowledge	Inexperience of attacker, not knowing relevant aspects
Lack of portability	Fact that a tool or technique available in one context (e.g., platform) is not available in the context in which the attacker wants to deploy the tool or technique
Library / module	Partition of an application as defined in software engineering
Limit scope of attack	Identify regions in the software or in an execution trace where next attacks steps should focus on, thus reducing the size of the code or data or trace where the attacker needs to perform next attack steps, thus reducing the effort to invest in them
Limit scope of attack by static meta info	Use standard information available in binaries (e.g., exported symbols) to identify regions in software or traces where next attack steps should focus on
Limitations from operating system	Properties of specific operating system on which the code has to be executed, that limit the attacker's capabilities in some way
Main()	Top-level function in an application

Make hypothesis	Based on the results of previous attack steps and background knowledge make a hypothesis about the features of an asset, protection, piece of software under attack, or attack tool capability that, if true, will enable certain attack steps to be performed successfully
Make hypothesis on protection	Make a hypothesis regarding the potential deployment or nature or features of a certain protection
Make hypothesis on reasons for attack failure	Make a hypothesis based on the observation that an previous attack step yielded insufficient results
Manually assist the disassembler	Interact with a disassembler tool to correct and complement its automated disassembler analyses
Memory access	The operation of reading or writing to main memory
Memory dump	Making a snapshot of (parts of) the code and data in the address space of a running program and dumping that image on disk for later analysis
Meta info	Standard information available in binaries, in the form of data that is not used by the software itself, but by the OS to load and launch the software correctly
Monitor public interfaces	Observing and analysing interaction (invocations, data passing, communications) between components along publicly available interfaces (such as exported functions in libraries or system calls)
Non-standard virtual machines	Customized virtual machines embedded in a protected program that interprets bytecode (in a custom, non-standard bytecode format) that replaces the original native code, thus hiding the semantics of the original code
Obfuscation	Form of software protection that increases apparent complexity of code or data
Obstacle	Feature or artefact that hinders attack steps and that is deliberatively put in place (or, if already present a priori, considered relevant for providing protection) by the defender in protected software, including in components of the software itself or of its execution environment
Obtain clear code after code decryption at run time	Identify and extract decrypted code in memory space of a running application under attack
Operation	Software functionality at the lowest level of granularity / abstraction
Out of context execution	Execute code fragments not as they are normally executed within the full program's execution, but in other crafted contexts (such as self-written main binary)
Overcome protection	Leaving a protection present and (partially) active, but manipulating the code and execution of the software or fragments therein such that the goal of an attack step is reached despite the protection still being present and (partially) active. This typically requires the custom, targeted development of external scripts and software components. The resulting code or execution are not necessarily representative for the original software as a whole, but they suffice for the attacker to reach his goal.
Pattern matching	Identifying code or data fragments of interest by comparing candidates to known patterns
Prepare the environment	Set-up and configuration of environment to execute and/or attack the program
Process / parent-child relation	Process are instances of software executing on a computer; parent processes launch child processes
Program input and output	External data consumed and produced by a program
Profiler	Tool used to collect statistics about execution of software elements
Profiling	Collecting statistics on a program's execution and its components (functions, instructions, libraries, ...)
Protection	Software protection technique applied on software under attack
Recognizable library	Part of software under attack that corresponds to a known library and is identified as such by the attacker

Recognize anomalous / unexpected behavior	Observe program features that contradict hypothesis about normal behavior given background knowledge of the attacker
Recognize similarity with already analysed protected application	Recognize that parts of a new piece of software under attack, although it maybe has been protected differently, is identical to parts of another software already attacked before, such that the knowledge of the already attacked version can be reused, thus reducing the amount of reverse-engineering effort needed
Recreate protection in the small	Create small program containing protection mock-up to aid in the development of attacks on that protection
Reference to API function / imported and exported function	Symbolic description of externally visible (standard library) functions provided by or needed by libraries and modules
Register	A component inside a central processing unit for storing information, that can be addressed directly in assembly code
Remote attestation	Online protection technique in which a secure server demands a running client to provide attestations to verify the integrity of the client
Replace API functions with reimplementation	Use of hooking, interposers, detours and other techniques to intervene in execution when external functions are invoked
Reuse attack strategy that worked in the past	Use background knowledge on paths of least resistance and successful attack paths based on attacks on similar pieces of software or on pieces of software protected with identical or similar (assumed or identified as such) protections
Round / repetition / loop	Specific instance of a program fragment execution in a trace containing multiple subsequent executions of the fragment; strongly connected component in a control flow graph
Run analysis	Invoke an automated analysis in a tool
Run software in emulator	Use emulation to execute software and to execute dynamic attacks
Size	Amount of code or data considered by an attacker
Socket	Data structure and its interface serving as an internal endpoint for sending and receiving data over a network
Software element	Aspects of a program of interest to an attacker
Static analysis	Analysis techniques that do not require code to be executed
Statistical analysis	Use of statistical techniques to identify and/or recover operations or data or features of interest
stderr	Output connection through which many programs output error messages
String	Sequence of alphanumeric text or other symbols in memory or an executable file
String / name analysis	Extracting information from names of files, exported functions, strings referenced in code fragments, etc.
Switch statement	Control flow structure resembling a `switch () { case ...: case ...: }` structure in C code
Symbolic execution / SMT solving	Determining (semantic) properties of code fragments using symbolic execution and SMT solving techniques
System call	The operation invoking system routines from the operating system
Tamper detection	Forms of software protection that try to detect that normal execution or code has been modified
Tamper with code statically	Edit code in the binary, e.g., to implement a workaround
Tamper with data	Alter data during the execution of a program
Tamper with execution	Alter ongoing execution by altering code or data
Tamper with execution environment	Adapt software or hardware in execution environment in ways not intended by their developers such that it supports specific attack tasks
Tool	Any software or hardware aid that automates activities needed in attack steps or that performs a task (semi)automatically
Tool limitations	Practical limitations (supported file sizes, memory consumption, lack of precision, ...) of a tool that make it unfit for the specific way an attacker wants to use it
Trace	Sequence of executed code fragments with or without additional properties of their execution

Tracer	Tool used to collect sequences of executed software elements and attributes of their execution
Tracing	Collecting a sequence of activities occurring during the execution of a program (instructions being executed, system calls, library calls, etc.)
Understand code logic	Act of reasoning about a code fragment
Understand persistent storage / file / socket	Act of reasoning about overall program behavior regarding storage - files - sockets
Undo the protection	Reversing the effect of a protection by undoing its deployment, i.e., reverting to the software without the protection (e.g., deobfuscating code or removing code guards).
Weak crypto	Use of cryptographic techniques that are even weak against black-box attacks
White box cryptography	Form of cryptography where keys do not occur in plain sight during execution
Workaround of a difficulty	Adaptation of tool or new tool that overcomes the limitation of existing attack tool; of a protection: see build workaround
Write tool supported script	Customize a tool using its built-in scripting features
XOR operation	The operation of performing an XOR on two or more data values; these occur very frequently in and around cryptographic primitives

References

Abrath B, Coppens B, Volckaert S, Wijnant J, De Sutter B (2016) Tightly-coupled self-debugging software protection. In: Proceedings of the 6th workshop on software security, protection, and reverse engineering (SSPREW), pp 7:1–7:10

Anckaert B, Madou M, De Sutter B, De Bus B, De Bosschere K, Preneel B (2007) Program obfuscation: a quantitative approach. In: Proceedings of ACM workshop on quality of protection, pp 15–20

Andrews AA, Ghosh S, Choi EM (2002) A model for understanding software components. In: 18th international conference on software maintenance (ICSM 2002), maintaining distributed heterogeneous systems, 3–6 October 2002, Montreal, Quebec, Canada, p 359

Armknecht F, Sadeghi AR, Schulz S, Wachsmann C (2013) A security framework for the analysis and design of software attestation. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security, CCS '13. ACM, New York, USA, pp 1–12. https://doi.org/10.1145/2508859.2516650

Barak B, Goldreich O, Impagliazzo R, Rudich S, Sahai A, Vadhan S, Yang K (2001) On the (im) possibility of obfuscating programs. Lect Notes Comput Sci 2139:19–23

Burkhardt J, Détienne F, Wiedenbeck S (2002) Object-oriented program comprehension: effect of expertise, task and phase. Empir Softw Eng 7(2):115–156

Cabutto A, Falcarin P, Abrath B, Coppens B, De Sutter B (2015) Software protection with code mobility. In: Proceedings of the second ACM workshop on moving target defense (MTD), pp 95–103

Capiluppi A, Falcarin P, Boldyreff C (2012) Code defactoring: evaluating the effectiveness of java obfuscations. In: 2012 19th working conference on reverse engineering (WCRE). IEEE, pp 71–80

Ceccato M, Capiluppi A, Falcarin P, Boldyreff C (2015) A large study on the effect of code obfuscation on the quality of java code. Empir Softw Eng 20(6):1486–1524

Ceccato M, Di Penta M, Falcarin P, Ricca F, Torchiano M, Tonella P (2014) A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empir Softw Eng 19(4):1040–1074

Ceccato M, Di Penta M, Nagra J, Falcarin P, Ricca F, Torchiano M, Tonella P (2009) The effectiveness of source code obfuscation: an experimental assessment. In: IEEE 17th international conference on program comprehension (ICPC), pp 178–187. https://doi.org/10.1109/ICPC.2009.5090041

Ceccato M, Dalla Preda M, Nagra J, Collberg C, Tonella P (2007) Barrier slicing for remote software trusting. In: Proceedings of the seventh IEEE international working conference on source code analysis and manipulation, SCAM '07. IEEE Computer Society, Washington, pp 27–36. https://doi.org/10.1109/SCAM.2007.6

Ceccato M, Tonella P, Basile C, Coppens B, De Sutter B, Falcarin P, Torchiano M (2017) How professional hackers understand protected code while performing attack tasks. In: Proceedings of the 25th international conference on program comprehension (ICPC), pp 154–164. ICPC best paper award and ACM Distinguished paper award

Collberg C, Thomborson C, Low D (1997) A taxonomy of obfuscating transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland

Collberg C, Nagra J (2009) Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection. Addison-wesley, London

Demissie BF, Ceccato M, Tiella R (2015) Assessment of data obfuscation with residue number coding. In: Proceedings of the 1st international workshop on software protection, SPRO '15. IEEE Press, Piscataway, pp 38–44. http://dl.acm.org/citation.cfm?id=2821429.2821440

Edmundson A, Holtkamp B, Rivera E, Finifter M, Mettler A, Wagner D (2013) An empirical study on the effectiveness of security code review. In: International symposium on engineering secure software and systems. Springer, pp 197–212

Flick U (2009) An introduction to qualitative research, 4th edn. Sage, London, England, UK

Glaser BG, Strauss AL (1967) The discovery of grounded theory. Aldine, Chicago, IL, USA

Katipally R, Yang L, Liu A (2011) Attacker behavior analysis in multi-stage attack detection system. In: Proceedings of the seventh annual workshop on cyber security and information intelligence research, CSIIRW '11. ACM, New York, pp 63:1–63:1. https://doi.org/10.1145/2179298.2179369

Letovsky S (1987) Cognitive processes in program comprehension. J Syst Softw 7(4):325–339

Linn C, Debray S (2003) Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of ACM conference on computer and communications security, pp 290–299

Littman DC, Pinto J, Letovsky S, Soloway E (1987) Mental models and software maintenance. J Syst Softw 7(4):341–355

Mallikarjunan KN, Prabavathy S, Sundarakantham K, Shalinie SM (2015) Model for cyber attacker behavioral analysis. In: 2015 IEEE workshop on computational intelligence: theories, applications and future directions (WCI), pp 1–4. https://doi.org/10.1109/WCI.2015.7495520

van Oorschot PC, Somayaji A, Wurster G (2005) Hardware-assisted circumvention of self-hashing software tamper resistance. IEEE Trans Dependable Secur Comput 2(2):82–92. https://doi.org/10.1109/TDSC.2005.24

Pennington N (1987) Stimulus structures and mental representations in expert comprehension of computer programs. Cogn Psychol 19(3):295–341

Piorkowski DJ, Fleming SD, Kwan I, Burnett MM, Scaffidi C, Bellamy RK, Jordahl J (2013) The whats and hows of programmers' foraging diets. In: Proceedings of the SIGCHI conference on human factors in computing systems. ACM, New York, pp 3063–3072

Prechelt L, Schmeisky H, Zieris F (2016) Quality experience: a grounded theory of successful agile projects without dedicated testers. In: Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, pp 1017–1027

Sillito J, Murphy GC, Volder KD (2006) Questions programmers ask during software evolution tasks. In: Proceedings of the 14th ACM SIGSOFT international symposium on foundations of software engineering, FSE, pp 23–34

Stol K, Ralph P, Fitzgerald B (2016) Grounded theory in software engineering research: a critical review and guidelines. In: Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016, pp 120–131

Strauss A, Corbin J (1990) Basics of qualitative research: grounded theory procedures and techniques. Sage, London

Sutherland I, Kalb GE, Blyth A, Mulley G (2006) An empirical examination of the reverse engineering process for binary files. Comput Secur 25(3):221–228

Udupa SK, Debray SK, Madou M (2005) Deobfuscation: reverse engineering obfuscated code. In: Proceedings of the 12th working conference on reverse engineering. IEEE Computer Society, Washington, pp 45–54. https://doi.org/10.1109/WCRE.2005.13. http://dl.acm.org/citation.cfm?id=1107841.1108171

Vectra (2017) Attacker behavior industry report. https://info.vectra.ai/attacker-behavior-industry-report-1q2017

Viticchié A, Regano L, Torchiano M, Basile C, Ceccato M, Tonella P, Tiella R (2016) Assessment of source code obfuscation techniques. In: Proceedings of the 16th IEEE international working conference on source code analysis and manipulation, pp 11–20

von Mayrhauser A, Vans AM (1994) Comprehension processes during large scale maintenance. In: Proceedings of the 16th international conference on software engineering, Sorrento, Italy, May 16–21, pp 39–48 von Mayrhauser A, Vans AM (1995) Industrial experience with an integrated code comprehension model. Softw Eng J 10(5):171–182

von Mayrhauser A, Vans AM (1996a) Identification of dynamic comprehension processes during large scale maintenance. IEEE Trans Softw Eng 22(6):424–437

von Mayrhauser A, Vans AM (1996b) On the role of hypotheses during opportunistic understanding while porting large scale code. In: 4th international workshop on program comprehension (WPC '96), March 29–31, 1996, Berlin, Germany, pp 68–77

von Mayrhauser A, Vans AM (1997a) Hypothesis-driven understanding processes during corrective maintenance of large scale software. In: 1997 international conference on software maintenance (ICSM '97), 1–3 October 1997, Bari, Italy, Proceedings, pp 12–20

von Mayrhauser A, Vans AM (1997b) Program understanding needs during corrective maintenance of large scale software. In: 21st intern. computer software and applications conference (COMPSAC'97), 1997, USA, pp 630–637

Wheeler DA (2001) More than a gigabuck: estimating gnu/Linux's size. https://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html

Wyseur B (2009) White-box cryptography. Ph.D. thesis, Katholieke Universiteit Leuven. https://www.cosic.esat.kuleuven.be/publications/thesis-152.pdf

Mariano Ceccato is tenured researcher in FBK (Fondazione Bruno Kessler) in Trento, Italy. He received the master degree in Software Engineering from the University of Padova, Italy, in 2003 and the PhD in Computer Science from the University of Trento in 2006 with the thesis "Migrating Object Oriented code to Aspect Oriented Programming". His research interests include security testing, migration of legacy systems, aspect oriented programming and empirical studies. He was program co-chair of the 12th IEEE Working Conference of Source Code Analysis and Manipulation (SCAM 2012) to be held in Riva del Garda, Italy.

Paolo Tonella is head of the Software Engineering Research Unit at Fondazione Bruno Kessler (FBK), in Trento, Italy. He is also Honorary Professor at University College London (UCL). He received his PhD degree in Software Engineering from the University of Padova, in 1999, with the thesis "Code Analysis in Support to Software Maintenance". In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications". He is the author of "Reverse Engineering of Object Oriented Code", Springer, 2005, and of "Evolutionary Testing of Classes", ISSTA 2004. Paolo Tonella was Program Chair of ICSM 2011 and ICPC 2007; General Chair of ISSTA 2010 and ICSM 2012. He is/was associate editor of TOSEM/TSE and he is in the editorial board of EMSE and JSEP. His current research interests include code analysis, web testing, search based test case generation and the test oracle problem.

Cataldo Basile received a M.Sc. (summa cum laude) in 2001 and a Ph.D. in Computer Engineering in 2005 from the Politecnico di Torino, where is currently a research associate. His research is concerned with software security, software attestation, policy-based security management in networked environments, policy refinement, and general models for detection, resolution and reconciliation of security policy conflicts.

Paolo Falcarin is a Reader in Computer Science at the University of East London, leading the Software Systems Engineering research group. He was awarded a PhD degree in Software Engineering from Polytechnic of Turin (Italy) in 2004. Paolo was the General Chair of SPRO 2015 and guest editor of a special issue on software protection of IEEE Software. His current research interests span over software engineering, distributed systems, software protection and security.

Marco Torchiano is an associate professor at the Control and Computer Engineering Dept. of Politecnico di Torino, Italy; he has been post-doctoral research fellow at Norwegian University of Science and Technology (NTNU), Norway. He received an MSc and a PhD in Computer Engineering from Politecnico di Torino. He is Senior Member of the IEEE and member of the software engineering committee of UNINFO (part of ISO/IEC JTC 1). He is author or co-author of over 140 research papers published in international journals and conferences, of the book 'Software Development–Case studies in Java' from Addison-Wesley, and co-editor of the book 'Developing Services for the Wireless Internet' from Springer. He recently was a visiting professor at Polytechnique Montréal studying software energy consumption. His current research interests are: green software, UI testing methods, open-data quality, and software modeling notations. The methodological approach he adopts is that of empirical software engineering. Contact him via email at marco.torchiano@polito.it. Follow him on twitter @mtorchiano

Bart Coppens is a postdoctoral researcher at Ghent University in the Computer Systems Lab. He received his Ph.D. in Computer Science Engineering from the Faculty of Engineering and Architecture at Ghent University in 2013. His research focuses on protecting software against different forms of attacks using compiler-based techniques and run-time techniques.

Bjorn De Sutter is professor at Ghent University in the Computer Systems Lab. He obtained his MSc and PhD degrees in Computer Science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and software protection. As for the latter, he developed techniques to mitigate reverse engineering, software tampering, code reuse attacks, fault injection attacks, and timing side channel attacks. He co-authored over 80 peer-reviewed papers in international conferences and journals. He coordinated the ASPIRE project.