Security Strength Measurement for Dongle-Protected Software

Many people consider dongles to be among the strongest forms of copy protection, but how much security do they actually offer? The model presented here aims to monetize the security strength of dongle-protected software by forecasting the amount of time a hypothetical attacker would take to break it.

Ugo Piazzalunga
Eutronsec Infosecurity

Paolo Salvaneschi
University of Bergamo

Francesco Balducci, Pablo Jacomuzzi, and Cristiano Moroncelli
Turin Polytechnic

One of the problems with software security is the lack of a proper methodology for guaranteeing a delivered security measure's efficacy. No governmental office will ever accept a new high-traffic river bridge, for example, without some evidence that the bridge can withstand the amount of traffic it's supposed to support. Yet, collecting evidence that a piece of software can sustain a defined security load seems to be a rare practice. One area in which such practice is indeed uncommon is software copy protection—specifically, copy protection with hardware dongles.

Although dongles are the top choice for software copy protection, they're typically found only in high-end, low-quantity software.[1] Nevertheless, IDC estimates that dongle-protected software generated a global revenue of US$117.8 million in 2003,[2] and a major vendor claims that the use of dongles has so far prevented $500 billion in software piracy (see www.aladdin.com/news/2005/HASP/aladdin_anniversary.asp). Despite these facts, though, little attention has been paid to the actual security that dongles offer.[1]

This article's objective is to develop a model for measuring the security strength of dongle-protected software. We believe such a measure is important because it can attach a clear, simple, and understandable monetization number to security (see www2.sims.berkeley.edu/resources/affiliates/workshops/econsecurity/econws/54.pdf and https://buildsecurityin.us-cert.gov/daisy/bsi/articles/bestpractices/architecture/10.html?branch=1&language=1).

Introduction to dongles

Dongles are USB keys or small boxes attached to the host parallel port and shipped to customers by software vendors along with the software they're meant to protect; the software will run only if it "finds" the dongle with which it was shipped.

As depicted in Figure 1, the dongle's logical resources include

- a symmetric encryption engine and storage for symmetric keys (public-key cryptography is rare; vendors used to include proprietary cryptographic algorithms, but today, they use standard ones);
- a persistent memory in which the software can read and write;
- a unique serial number;
- a unique independent software vendor identification number, which the vendor assigns to each of its dongle customers; and
- an access password to unlock the dongle's functionality.

However, not all of these resources are always present—for example, memory might not be available.

The copy-protected application interacts with the dongle and progresses its execution only if the dongle answers appropriately. The interaction between the software and the dongle takes place through calls to the dongle API; Figure 1 explicitly shows the API's core functions, `AES-encrypt` and `read-write-memory`.

Security is a system property: it must be addressed at all levels of abstractions and for all components and interactions. In this system, the macrocomponents are the dongle, the dongle-protected software, and the

Related work in security measurement

Sandmark,[1] a tool developed at the University of Arizona under the supervision of Christian Collberg, gave us the initial inspiration for this work. Our work concentrates on the evaluation aspects of security protection, but Sandmark includes only standard software engineering metrics. David Nicol[2] presents a sample of the models applied in security evaluation efforts; the attacker-centric approach he describes is similar to our proposal, the main differences being the statistical approach and the state model he uses as opposed to the deterministic approach and attack tree model we adopted. A few models appear in the area of networked systems: Stuart Schechter[3] applies the regression models to security risks; Vibhu Saujanya Sharma and Kishor Trivedi[4] use the discrete time Markov chains to model security risks based on the vulnerability knowledge of its components. Our work differs by going to a deeper level than vulnerabilities: we take into account the inner factors that might lead to vulnerabilities—that is, applied defenses and how to apply them. Mehmet Sahinoglu[5] describes a decision-tree model for quantifying risks. Similar to our model, the Sahinoglu approach also depends on countermeasures to known threats. However, unlike our model, in which we explicitly link the fine-grained details of countermeasures to attack efforts, the Sahinoglu model depends on a probabilistic lack-of-countermeasure input.

Sean Barnum and Gary McGraw describe the importance of knowledge catalogs that compile and share critical software security knowledge.[6] Our efforts follow their proposal by organizing and interrelating software security knowledge specific to dongle protection. James Whittaker and his colleagues present extensive and detailed attack catalogs, but the interrelation of the attacks to the other security facets that we address—defenses, measurements, knowledge-compiled automatic tools—is missing.[7] We go beyond Barnum and McGraw's proposal by defining a model that monetizes software security strength. (Due to space limitations, we report here only a limited description of attacks and defenses; we refer the reader to Francesco Balducci and his colleagues for an extensive and detailed presentation.[8])

Security evaluation is an instance of the more general issue of measuring software quality. Many different procedures for measuring the quality of software products are available in the literature. Despite these different approaches, we can identify several common components:

- a tree of attributes linking low-level measures to high-level abstractions;[9]
- an algorithm for generating values of high-level attributes from measures;[10]
- a process model (the deliverables of the development phases);[11] and
- a product model (a model of the software components to be measured).[11]

The attributes tree has been largely explored, from James A. McCall's original work to the ISO 9126 standard;[9] the types

continued on p. 34

host running it. The dongle itself is rarely the point of attack for hackers (see www.woodmann.com/crackz/Dongles.htm); rather, the weakest link, which is what this article focuses on, is the interaction between the software and the dongle API.

Further details on dongle technology and its strengths and limitations appear elsewhere.[3,4]

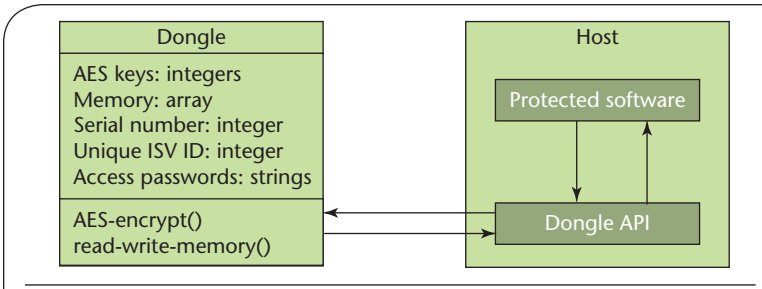

Figure 1. Dongle. This abstract representation shows the dongle's interaction with the software it's meant to protect.

Methodology

The methodology we used to develop a security strength metric of copy protection through dongles consists of several steps:

- Compile a *defense pattern catalog* (each defense pattern includes measurable attributes about the strength of defense provided); build an *attack pattern catalog* (by breaking attacks into activities that can be combined into full-blown attacks); and construct *experimental cost-to-break functions*[5] (these flow diagrams represent our experimental knowledge of how to conduct an attack in a catalog, the result of which is the measure in minutes of how long an attacker will need to successfully conduct the attack).
- Interrelate these security facets into a combined *attack tree model*. An attack tree is an AND/OR graph that links possible attacks; at the leaf nodes, this tree is augmented with the experimental cost-to-break functions. The tree supports the security strength computation.
- Execute the model. We compute the security strength metric by propagating through the attack tree the output values of the experimental cost-to-break functions. The resulting value is the estimated time required to break the existing defenses, given a tree of attacks and an implemented set of defenses

continued from p. 33

of measures have received great research interest as well,[11] and many variations of the algorithm for composing measurements are available.[10] We assume that we can execute the measurement procedure (with constraints) at each development phase, which in turn assumes the existence of a defined process model that delivers a suitable set of documents at the end of each phase.

Despite the large number of contributions, an integrated approach that evaluates a specific software quality by using product models, measurable attributes, inputs, and computed effects is uncommon. One exception to this general scarcity of models for evaluating the quality of software products is the approach presented in the Architectural Trade-off Analysis Method (ATAM) with Attribute-Based Architectural Styles (ABAS) and Tactics patterns for software architecture evaluation.[12] ATAM defines both a process and knowledge support for evaluating and improving the design of software architectures. The method is applicable to software products in general, is focused on the evaluation of software architectures, and is aimed mainly at providing qualitative suggestions to the designer. Our aim is to build quantitative models able to predict a specific quality value for a specific class of software products.

References

1. C. Collberg, G. Myles, and A. Huntwork, "Sandmark—A Tool for Software Protection Research," *IEEE Security & Privacy*, vol. 1, no. 4, 2003, pp. 40–49.
2. D.M. Nicol, "Modeling and Simulation in Security Evaluation," *IEEE Security & Privacy*, vol. 3, no. 5, 2005, pp. 71–74.
3. S.E. Schechter, "Toward Econometric Models of the Security Risk from Remote Attack," *IEEE Security & Privacy*, vol. 3, no. 1, 2005, pp. 40–44.
4. V.S. Sharma and K.S. Trivedi, "Architecture-Based Analysis of Performance, Reliability and Security of Software Systems," *Proc. 5th Int'l Workshop on Software and Performance*, ACM Press, 2005, pp. 217–227.
5. M. Sahinoglu, "Security Meter: A Practical Decision-Tree Model to Quantify Risk," *IEEE Security & Privacy*, vol. 3, no. 3, 2005, pp. 18–24.
6. S. Barnum and G. McGraw, "Knowledge for Software Security," *IEEE Security & Privacy*, vol. 3, no. 2, 2005, pp. 74–78.
7. J.A. Whittaker and H.H. Thompson, *How to Break Software Security*, Addison-Wesley, 2004.
8. F. Balducci, P. Jacomuzzi, and C. Moroncelli, *Security Measure of Protected Software: A Methodology and an Application to Dongles*, MS thesis (in Italian), Dept. of Electronics, Turin Polytechnic, 2005.
9. ISO/IEC 9126-1, "Information Technology—Software Product Quality—Part 1: Quality Model," 1999.
10. J. Voas, "Trusted Software's Holy Grail," *Software Quality J.*, vol. 11 no. 1, 2003, pp. 9–17.
11. R. Bache and G. Bazzana, *Software Metrics for Product Assessment*, McGraw-Hill, 1994.
12. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.

and their attributes. We compiled the catalogs as well as the experimental cost-to-break functions into software tools that can automatically measure some of the metrics of defense patterns and compute the software's security strength.
- Validate the model. The predicted security strengths should be consistent with field data, which could come either from historical information or through specific real attacks on a sample of security systems.

The following sections detail each step of the methodology, the catalogs, and the experimental cost-to-break functions in the specific context of our study.

Defense pattern catalog

John Viega and Gary McGraw describe the principles behind effective piracy protection in great detail.[3] We must consider these principles—scattering license management software, code obfuscation, checksums, and responding to misuse—for dongles as well. Table 1 shows our defense catalog, which includes these principles along with the additional dongle-specific principle "be sure to talk to the dongle," which seems trivial, but is a common software developer error. The first two defenses in Table 1 are based on this principle. For each attribute, we indicate if the developed tools can provide an automatic measure based on dynamic behavior or static analysis.

Note that protecting software by using only the specific defenses made possible by dongles will result in poor overall protection—for example, if we don't scatter our "talk" with the dongle, the hacker just has to find the few locations in the code in which the software interacts with the dongle and patch them. If we apply generic piracy-prevention defense strategies as well, the security strength we measure won't be the protection afforded by the dongle per se but the whole resulting protection.

We don't mention the "scattering license management" principle here because it's implicitly included in the first two defenses. A high number of caller locations means that the AES challenge-response or memory-usage defenses are implemented in multiple locations in the code; thus, the copy-protection scheme is well scattered.

Let's look closer at the defenses and their attributes. A detailed description appears elsewhere.[6]

AES challenge-response. AES encryption is the basis for a challenge-response protocol between the host software and the dongle. However, it makes no difference whether the protocol is based on AES or another strong cryptographic algorithm because the aim is to uniquely identify the key and verify that it's "real." The challenge must be unpredictable—perhaps generated with a secure random number generator.[7] A predictable

challenge would allow a dongle emulation attack.

The attributes of this defense are the number of different caller addresses (the different locations from which the executable program makes calls to the AES function), the challenge distribution (measured by making sure that no repetition exists), and the number of different AES keys. Using as many keys as possible requires attackers to discover them all.

Memory usage. The second defense is memory usage, which is also based on the principle "make sure we're talking to the real key." It's effective only against hardware- or kernel-level emulators because they generally record all the communication between the application and the dongle but don't try to understand the semantics. A countermeasure is to write a random number to memory and then read it (www.eutron infosecurity.info/pub/smartkey/kit/current/Docs/SmartKeyUserManual.pdf). If the numbers differ, the protected software is "talking" to an emulator.

Memory usage is also useful against fake libraries, but only if the memory content is set at the software vendor site, the software depends on correct data to be read out of memory, and the hacker has no access to the key or its content. Here, the technique is to avoid any explicit control of the data read—control is implicit in the sense that the program wouldn't work correctly otherwise (for example, we could store the values of numeric constants or a function's binary code to be called later). It's critical to scatter this defense as much as possible. This property is measured through the number of different caller addresses.

Code obfuscation. Malware writers often deploy obfuscation techniques to protect their programs from analysis.[8] Similarly, obfuscation can enhance the protection of software through dongles. A simple way to complicate the static analysis of protected code is to compress its executable,[4] but an increased level of sophistication is needed against program analysis through a debugger. Anti-debugging techniques[1,4] could partly prevent the use of debuggers against a protected application. Christian Collberg and Clark Thomborson proposed several methods to transform code in such a way so that its semantic is preserved but it becomes hard for humans to understand.[1] The attribute we listed in Table 1 for this defense is simple and automatically measurable: the arguments passed to the API functions are searched in the executable file, and if found, they aren't obfuscated.

Static link to dongle library. The protected software must be linked either statically or dynamically against the dongle library. A dynamic link exposes a clear separation between the protected software and the library: an attacker can simply replace the library with a fake one.

Table 1. The defense pattern catalog and associated attributes.

DEFENSE PATTERNS AND ATTRIBUTES	MEASUREMENT TECHNIQUES
AES challenge-response	
Number of different caller addresses	Tool based: dynamic
Challenge distribution	Tool based: dynamic
Number of AES keys	Tool based: dynamic
Memory usage	
Number of different caller addresses	Tool based: dynamic
Static link to dongle library	
Link type	Tool based: static
Code obfuscation	
Obfuscated arguments in calls to dongle API	Tool based: dynamic + static
Proper error handling	
Distance of check from error handling	Manual
Distance of library call from error handling	Manual
Obfuscated error messages	Manual
Function used to show error message	Manual
Time message remain in clear at runtime	Manual
Integrity checks	
Existence of runtime checksum	Manual
Existence of static checksum	Manual

Proper error handling. A trivial management of errors might provide attackers with easy hints on how to progress in their attacks. In general, the location of the code managing the misuse should be separated from both the library call's location and the location of the code checking the call's outcome. Error messages that could help attackers should be avoided or at least encrypted. Viega and McGraw suggest using decoys to misdirect the attacker's analysis and offer techniques to respond to misuse by causing hard-to-trace program misbehaviors.[3]

The first three attributes of this defense measure how far away the library calls and the code checking their outcomes are from error handling and whether the error messages are obfuscated. We also introduce two additional attributes: "function used to show error message" and "time message remains in clear at runtime." When messages are obfuscated, an attacker might try to trace calls to standard print functions such as C's `printf`. The longer the message remains deciphered in RAM, the greater the chance for the hacker to find it.[9]

Integrity checks. Software should have ways to check its own integrity and the integrity of libraries on which it depends. If the integrity check fails, it could

Table 2. The attack pattern catalog with steps for each attack.

ATTACK PATTERNS	STEPS
Remove checks (A)	1. Locate checks starting from error messages 2. Modify code with hexadecimal editor
Remove checks (B)	1. Locate checks starting from library calls 2. Modify code with hexadecimal editor
Memory tampering	1. Locate a library call 2. Debug to capture the clear password 3. Analyze and modify memory content
Patch at runtime (A)	1. Locate checks starting from error messages 2. Write script for patcher
Patch at runtime (B)	1. Locate checks starting from library calls 2. Write script for patcher
Emulate dongle	1. Debug to capture the AES keys 2. Develop the code for emulation library 3. Deviate calls of the original library to calls of the emulation library

be because someone tampered with the code. Runtime patching of executable code makes checksums on static files ineffective; checksums, therefore, should be executed on code in memory.[3] If possible, avoid direct comparisons of the computed checksums with the correct values because this exposes the expected correct value (for example, the checksum could be used as a key to decrypt the code that runs afterward).

Attack pattern catalog

We assume the "malicious host attack" view of security here: we're considering attacks to "benign" software installed on a "malicious" host, which fall into four main categories:

- Reverse-engineering attacks[10] deploy tools such as disassemblers to dig into protected code and unprotect it.
- Fake library attacks try to divert protected software into believing it's talking to the real dongle while it's actually interacting with a software emulator.
- Memory-tampering attacks try to alter the dongle's memory content to circumvent some parts of its software protection. The memory could contain critical data such as the software license expiration date.
- Kernel-level emulator attacks record and replay all the communication between the application and the dongle. Due to the stricter security of recent Microsoft operating systems, this kind of attack seems to have disappeared, so we won't consider it further here.

Table 2 lists attack patterns and defines the steps comprising each attack (the subattacks).[6] The term *check* refers to instructions in the software's executable file that implement copy-protection mechanisms.

Remove checks (A) and (B). Attackers might have collected any errors raised while executing the software; by using a disassembler, they can search for them and find exactly where messages are referenced. The lines with referenced messages are the starting points for hunting checks. Often, compare or test instructions followed by jumps in the vicinity of these lines are the signature used to locate checks.

The previous step, if successful, ends by crafting (through a hexadecimal editor) the changes needed to circumvent the checks. Remove checks (B) is identical to remove checks (A) except for the strategy used to locate the checks: instead of using the error to trace back the check, it uses calls to the dongle library.

Patch at runtime (A) and (B). Patch at runtime (A) and (B) differs from remove checks (A) and (B) in the way the executable code is modified. Attackers can use a patcher that loads the unmodified code and, before executing it, applies the specified changes to the already loaded code.

Memory tampering. Memory tampering might be used to attack a protection policy by forcing alteration of the dongle's memory. Software vendors often use the memory to hold critical protection-related data. A debugger can easily discover the dongle's password, which potentially enables the attacker to change memory content or duplicate the dongle. This attack assumes the most favorable scenario for the attacker: access to both the copy-protected software and the dongle.

Emulate dongle. The attacker's objective is to develop a software library that emulates dongle behavior. The signatures of the functions to be simulated are accessible in the dongle vendor's documentation; an array with read–write operations can emulate the dongle's memory functionality. To discover the dongle's initial content, attackers can either debug to capture the dongle's passwords and use the vendor's tools to access the data or run the emulator in "registration mode."

Registration mode makes it possible to emulate memory functionality and analyze the way calls are made—for example, if the AES challenge sent to the dongle isn't chosen randomly, the attacker can simply implement a challenge-response table built from the logs. If challenges are randomly generated, the attacker must discover the AES keys via reverse engineering (a more time-consuming process). Obfuscated code[1] and obfuscated cipher[11] significantly increase the discovery time. Assuming hackers have coded the fake library, they still need to divert the original library calls to the fake one. The difficulty of this step depends on how the protected software links to the dongle library (dynamically or statically).

Experimental cost-to-break functions

The cost-to-break function's goal isn't to provide a precise, comprehensive measure of the cost to break a dongle—rather, it's meant as a first approximation of a reasonably realistic time effort that a hacker with regular tools and resources needs to conduct a specific attack. In most cases of dongle-protected software, this is indeed appropriate. *First approximation* implies that as we gather more experience and knowledge about attacks, we can improve our estimates (and the model).

Each function embeds empirical knowledge and estimates of the time effort required to perform a specific attack step. Function inputs are values assigned to defense pattern attributes. Each function is represented by a flow diagram that assigns and modifies the value of the time effort to execute the attack. The flow diagram represents our knowledge of how to conduct an attack as well as our experimental knowledge of how difficult it is to perform a certain attack step, depending on defense attributes.

In Figure 2, green boxes are function inputs, which include attributes of the defense patterns' memory usage and proper error handling. These attributes' values are used either as the base of conditional decisions or as direct values for the cost-to-break assignment. Conditions always include a "default" branch (the easiest for the attacker) to be taken whenever the attribute value is unavailable.

We can interpret Figure 2 as follows. If error messages aren't obfuscated, it's trivial to find them in the executable code. Furthermore, if the error messages are close to the code in which the copy-protection logic is implemented, the attacker can counterfeit it. The total cost of locating all the code in which this logic is implemented is proportional to the number of checks in the code. Note, too, the assignment of the *td* value in the figure: in this case, as we gather through experience a better understanding of the relationship between the attack effort and the distance of the check code from the error message, we could provide a more detailed flow to capture this relationship without needing to manually assign a value.

What if the defender obscures error messages? With the help of a debugger, attackers will be able to find where messages are de-obfuscated, but it will cost them 12 minutes. Where does this magic number come from? We set up a battery of test programs that implemented the defense patterns presented earlier and varied their attributes. We had 13 different executable test programs and applied a total of 55 attacks to them. More specifically, three of us (the three authors from Turin Polytechnic) acted as hackers and performed the attacks. We then measured the time each hacker needed to perform the attacks: the magic numbers here are the average times measured during these attack sessions.

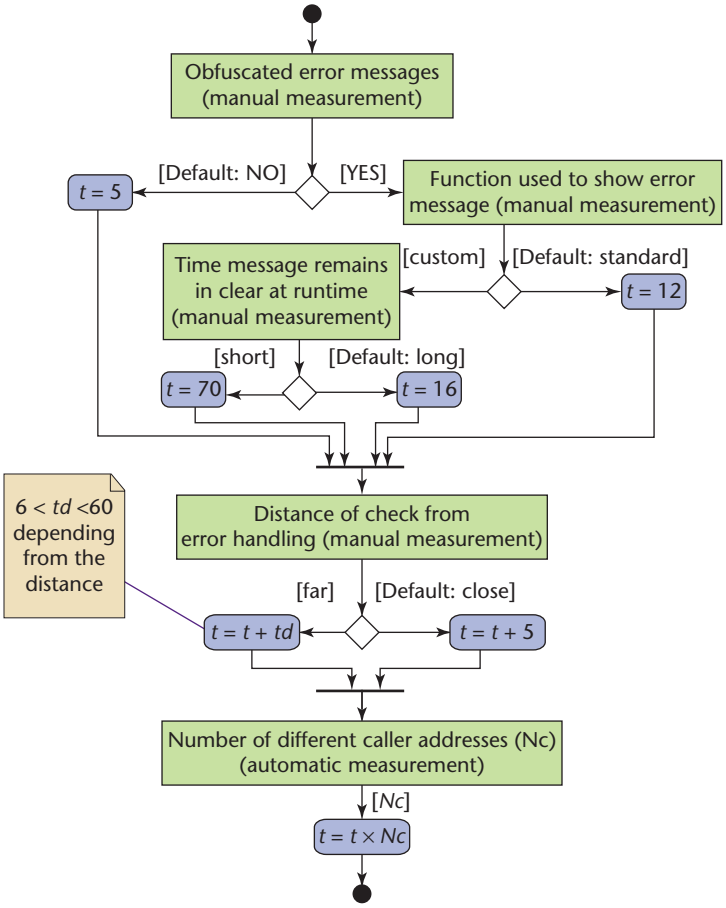

Figure 2. Experimental cost-to-break function for the subattack "locate the checks starting from error messages." *t* is the time effort in minutes to execute the attack. The green boxes are function inputs.

Some of the numbers, such as the 70 assigned when the time message remains shortly in clear at runtime, are arbitrarily assigned a reasonable value because we didn't have data coming from this session.

We limit the presentation of the experimental cost-to-break functions to this attack example. A complete presentation appears elsewhere.[6]

Attack tree model

Our security model is based on attack trees.[12] We extended the attack tree model by augmenting each leaf node with its corresponding experimental cost-to-break function. Figure 3 shows the tree for our reference software.

Given the augmented attack tree, we then collect all the values to be assigned to the defense pattern attributes. Without manual inspection, we can run our extraction tool on some of these values and assign to the leaf nodes the output values of the experimental cost-to-break functions. The leaf values are propagated upward using the rule "OR nodes take the value of their cheapest child; AND nodes take the value of

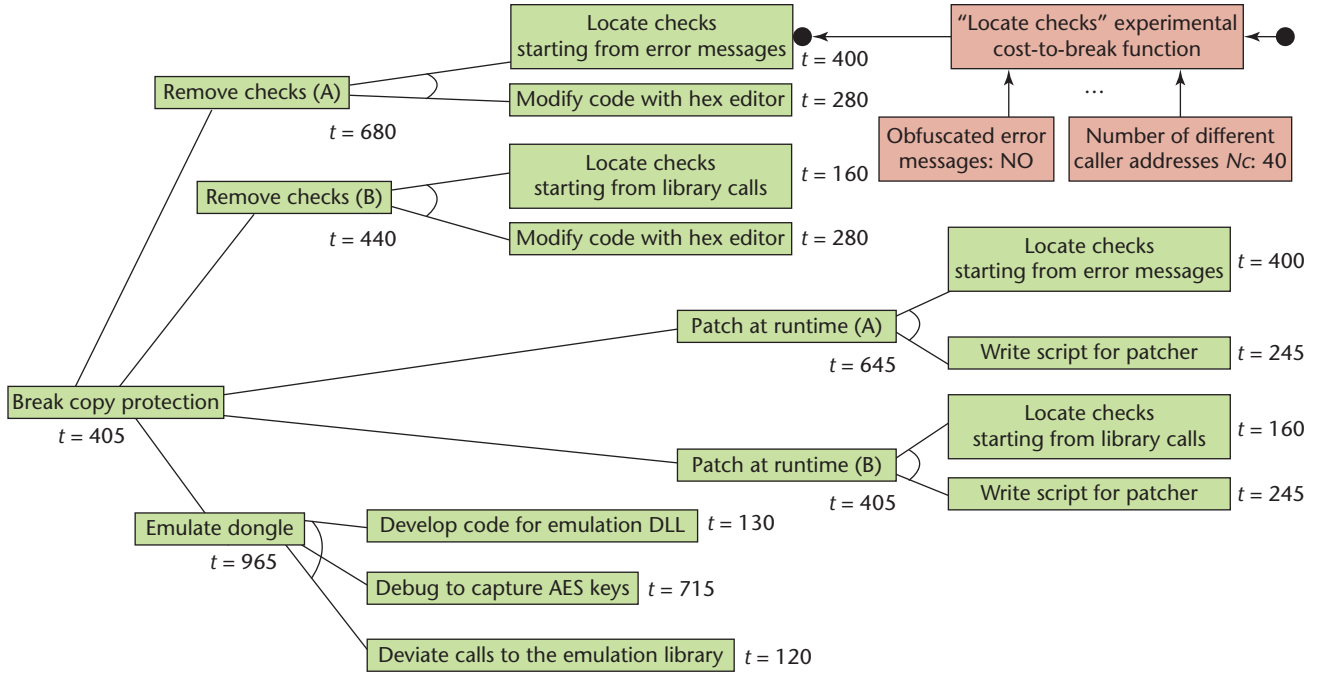

Figure 3. Augmented attack tree. For space reasons, only the first top leaf node is shown augmented with its associated experimental cost-to-break function (*t* is the time effort in minutes to execute the attack). The *t* value of the other leaf nodes is the output of the experimental cost-to-break function associated with each node, given the attribute values of defense patterns on which they depend.

the sum of their children" (AND nodes are connected through an arch).[12] For the software in Figure 3, the security strength computed this way is 405, which means a hypothetical attacker would need 405 minutes to unprotect it, the "patch at runtime (B)" attack being the cheapest child of the tree's root node.

We built tools for measuring some of the defense pattern attributes and computing the security strength:

- The *logger* is meant to be hooked to the software under analysis and is implemented through a "fake" library in registration mode. After the registration sessions, we use the log file (detailing all the calls made) as input to the analyzer.
- The *analyzer* assigns values to attributes. Table 1 lists all the metrics defined for the defense patterns and whether the analyzer can measure them. The log mentioned earlier is used to infer values that can be derived from the software's dynamic behavior. The values of attributes measured through static analysis are obtained by inspecting the binary files.
- Given an assignment of the defense pattern attribute values, our *security strength computation tool* computes the security strength estimation as the time a hacker would need to break the software.

We can learn many things from our tree model. Unlike other attack costs, for example, the "emulate dongle" cost-to-break function doesn't depend on how much the AES challenge-response and memory usage defenses are scattered, so it establishes an upper limit on the security strength of 965 minutes. Even if we kept increasing the number of different caller addresses (*Nc*), we would reach a point at which this increase becomes worthless (for this software, 96 caller addresses). Defenders therefore have to take into account an appropriate balance among the defenses adopted—for example, if we seek strength greater than 965 minutes, obfuscation must be used to strengthen the AES pattern.

Suppose we want to achieve a strength of 10,000 minutes (roughly 20 worker-days). What are the combinations of defense patterns we should implement? Through what-if experiments, we can change the defense patterns and attributes and compute the resulting security strength. Combinations for 10,000-minute strength would require obfuscation to protect the AES defense pattern. Among these combinations, we have one in which we could raise *Nc* to have a value near 1,000, but it seems time-consuming to implement so many checks in the code. Is there a more effective way? In a word, yes—we could expand the model to include the costs of implementing defenses and support the search of the best cost–benefit combination.

Field validation

To validate the methodology, we gathered five Italian

Table 3. Values assigned to defense pattern attributes for analyzed software.

DEFENSE PATTERNS AND ATTRIBUTES	S1	S2	S3	S4	S5	T
AES challenge-response						
Number of different caller addresses	1	0	1	0	1	40
Random challenge distribution	no	n/a	no	n/a	no	yes
Number of AES keys	n/a	n/a	n/a	n/a	n/a	20
Memory usage						
Number of different caller addresses	2	1	1	2	1	0
Static link to dongle library						
Link type	dyn	dyn	static	static	static	static
Code obfuscation						
Obfuscated arguments of calls to dongle API	yes	yes	yes	yes	yes	yes
Estimated security strength (min)	**53**	**22**	**49**	**39**	**66**	**405**
Actual cracking time (min)	**70**	**35**	**65**	**50**	**80**	**320**

commercial software packages protected by Eutronsec's dongles (from S1 to S5 in Table 3). They aren't detailed further because of privacy reasons. We used only binaries and dongles as input for this validation, but we considered additional test software (T in Table 3) that we developed as a reference point. This reference software implements only the AES pattern—the attacker is supposed to have the dongle and can therefore easily discover its memory content. The last two rows in the table report the estimated security strength and the actual cracking time in minutes.

For each piece of software, we logged the data through the logger, measured the values for the defense pattern attributes through the analyzer, computed the security strength, and hacked the software and recorded the time needed to crack it (see Table 3).

It was amazingly easy to crack all five commercial software applications. The estimated security strengths reflect this by assigning values comparable to measured cracking times.

Although the evidence in Table 3 is insufficient to claim that our proposed model provides a good model of the security strength in dongle-protected software, we believe these initial results are encouraging. A further indication of our model's value is that it moves the ability to reason about security into a domain that can be numerically ranked—for example, we can prove that 965 minutes is the upper bound for the security strength of software that doesn't use obfuscated AES.

Yet, we experienced some limitations:

- We had low variability in the complexity of the implemented copy-protection schemes, as all the five software have low security strength. However, the reference test software seems to indicate that the approach can pick up increasing levels of protection.
- The sample of software analyzed might be too small (it wasn't possible to gather additional commercial software).
- We considered only a minimal set of defense attributes; more of them should be added (for example, anti-debugging techniques for the code obfuscation defense).

We could argue that the poor protection we observed was due to vendor-specific issues (Eutronsec's dongles might have a poor baseline level of security, for example). Although we can argue that this is unlikely because the architecture for dongle protection is identical among most vendors, we performed attacks on two commercial software applications sold worldwide from US (U) and European (E) companies. Both were protected by dongles from another major vendor, but software U was easily cracked. Software E resisted our one-day effort; our reverse-engineering analysis discovered that the defenses presented here were implemented in the proper way.

The steps in our approach are generic, so our methodology could also apply to other security fields, including Web applications.[13,14] It requires explicitly defined attributes that can be associated to Web application defenses patterns (such as sanitized inputs, idle time for session time-out, and delay between users submitting login credentials and success or failure response). We would then need to link together defense and attack patterns by building the experimental cost-to-break functions and the augmented attack tree models. Comprehensive tools for automatic metric estimation are probably difficult to build; you could start with the simpler-to-implement attribute measurements and involve architects or developers to estimate the more complex attributes.

Our field validation seems to confirm the hacker belief that "software developers who use [dongles] usually aren't [...] motivated [...], they assume that because they are using a dongle that will be de-

terrent enough [...]" (www.woodmann.com/crackz/Dongles.htm). We shouldn't just blame software developers, though—if we want better security, dongle technology must be usable in a secure way. The models and tools described here address this responsibility. Before security testing, our methodology lets developers rank strength; during development, it supports what-if experiments, to help predict strength with a particular combination of defenses.

Further progress remains to be done. We envisage an extension to our current model that identifies the easiest-to-implement combination of defense patterns and the associated attributes to reach it, given the desired software strength. We've speculated that our methodology to monetize security strength might be applied to other realms, such as Web applications. Showing that this is indeed possible is another future challenge. ☐

Acknowledgments

We thank Chintan Vaishnav from MIT for his corrections and suggestions. Many thanks to the anonymous reviewers for their helpful comments.

References

1. C.S. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection," *IEEE Trans. Software Eng.*, vol. 28, no. 8, 2002, pp. 735–746.
2. *Worldwide Hardware Authentication Token 2004-2008 Forecast and 2003 Vendor Shares*, IDC document #31432, IDC, June 2004.
3. J. Viega and G. McGraw, *Building Secure Software*, Addison-Wesley, 2001.
4. P. Cerven, *Crackproof Your Software*, No Starch Press, 2002.
5. S.E. Schechter, "Toward Econometric Models of the Security Risk from Remote Attack," *IEEE Security & Privacy*, vol. 3, no. 1, 2005, pp. 40–44.
6. F. Balducci, P. Jacomuzzi, and C. Moroncelli, *Security Measure of Protected Software: A Methodology and an Application to Dongles*, MS thesis (in Italian), Dept. of Electronics, Turin Polytechnic, 2005.
7. R. Anderson, *Security Engineering*, Wiley, 2001.
8. S. Ring and E. Cole, "Taking a Lesson from Stealthy Rootkits," *IEEE Security & Privacy*, vol. 2, no. 4, 2004, pp. 38–45.
9. M. Howard and D. LeBlanc, *Writing Secure Code*, Microsoft Press, 2003.
10. G. Hoglund and G. McGraw, *Exploiting Software*, Addison-Wesley, 2004.
11. S. Chow et al., "A White-Box DES Implementation for DRM Applications," ACM CCS-9 DRM Workshop, 2002
12. B. Schneier, "Attack Trees: Modeling Security Threats," *Dr. Dobb's J.*, vol. 24, no. 12, 1999, pp. 21–29.
13. M. Andrews and J.A. Whittaker, *How to Break Web Software*, Addison-Wesley, 2006.
14. Open Web Application Security Project Foundation, "A Guide to Building Secure Web Applications and Web Services, 2.1" (DRAFT 3), Feb. 2006; www.owasp.org/index.php/Category:OWASP_Guide_Project.

Ugo Piazzalunga *is director of R&D at Eutronsec. His technical interests include software security and the design of usable security devices. Piazzalunga is a member of the ACM and the IEEE Computer Society. Contact him at upiazzalunga@acm.org.*

Paolo Salvaneschi *is an associate professor of software engineering at the University of Bergamo, faculty of engineering. He joined the university after serving 20 years as a software architect and head of software development groups in industry and research labs. Salvaneschi's interests include quality evaluation of software products, software design methods, and artificial-intelligence-based data interpretation. Contact him at pasalvan@alice.it.*

Francesco Balducci *is a consultant for ST Microelectronics on Linux-embedded systems and DSP processors. Previous research involved software security and hardware dongles at Eutronsec Infosecurity and IT solutions for enterprises at Reply. Balducci has a degree in electronic engineering from Turin Polytechnic. Contact him at francescobal@msn.com.*

Pablo Jacomuzzi *is a consultant of hardware and software-embedded developments at Reply. His technical interests include hardware and software developments, especially in the embedded environment, and software security. Jacomuzzi has a degree in electronic engineering from Turin Polytechnic. Contact him at pablo@jacomuzzi.it.*

Cristiano Moroncelli *is a consultant for ST Microelectronics on Linux-embedded software development and DSP programming. He has a degree in electronic engineering from Turin Polytechnic. Contact him at moroncellicristiano@libero.it.*