Improving Fuzz Testing Using Game Theory

Conference Paper · September 2010

DOI: 10.1109/NSS.2010.81 · Source: DBLP

5 authors, including:

Jorge Lucángeli Obes

8 PUBLICATIONS **84** CITATIONS

SEE PROFILE

Radu State

University of Luxembourg

329 PUBLICATIONS **2,370** CITATIONS

SEE PROFILE

2010 Fourth International Conference on Network and System Security

Improving Fuzz Testing using Game Theory

Sheila Becker*, Humberto Abdelnur†, Jorge Lucángeli Obes‡, Radu State* and Olivier Festor†
*University of Luxembourg, Luxembourg
Email: {sheila.becker, radu.state, thomas.engel}@uni.lu
†MADYNES - INRIA Nancy-Grand Est, France
Email: humberto.abdelnur@loria.fr
‡Universidad de Buenos Aires, Argentina
Email:jlucangeli@dc.uba.ar

***Abstract*—We propose a game theoretical model for fuzz testing, consisting in generating unexpected input to search for software vulnerabilities. As of today, no performance guarantees or assessment frameworks for fuzzing exist. Our paper addresses these issues and describes a simple model that can be used to assess and identify optimal fuzzing strategies, by leveraging game theory. In this context, payoff functions are obtained using a tainted data analysis and instrumentation of a target application to assess the impact of different fuzzing strategies.**

I. Introduction

Fuzz testing has emerged as one major approach to rapidly identify vulnerabilities in applications and embedded devices. The idea behind fuzz testing is to inject unexpected input into an application with the objective to find particular input that triggers a vulnerability. The potential input space can be huge, assuming one application that accepts one single input represented by one byte, we should fuzz test all the 256 possible different values. Testing has to be done and can become much more complicated if we consider additional constraints - like signed/unsigned integers and potential conversions. When the input space gets larger, the curse of dimensionality becomes even worse. One major research challenge for fuzz testing is to chose efficient search strategies including the development of metrics to quantify the efficiency of a fuzz test.

In this paper, we address a game theoretical model to determine the best strategy to be used, by investigating on the efficiency of several search strategies. This is done with respect to metrics that take into account the impact of a fuzz test on a target application. We measure this impact using a system level instrumentation that leverages tainted data analysis and information theoretical concepts. The paper is structured as follows: Section II gives an overview on our instrumentation framework. Section III describes the game models used to represent a fuzz test and shows the concrete instantiation of these models within a fuzzing framework. Experimental results are shown in Section IV and relevant related work given in Section V. We conclude the paper and future works are discussed in Section VI.

II. Instrumentation

We have implemented two tracing environments. The first one is specific to Linux implementations, while the second is targeted to MS-Windows architectures. In both cases, we were interested in obtaining the memory operations involved with fuzz-test generated data. We relied on tracking *tainted data* [5], where one considers all data coming from the network as tainted and then sees how the "taintedness" spreads as that data is processed. Approaches based on dynamic instrumentation techniques generate a considerable overhead [11], so we have developed a library-call tracing mechanism and concentrate on memory copy functions like `strcpy` or `memcpy` . The major issue we had on both architectures relates to the tracing of multi threaded applications. When tracing and debugging an application, breakpoints are set to control the flow. Once hit, breakpoints have to be disabled so that the program can complete execution. One option is to replace the breakpoint with the original instruction. This is tricky, because the breakpoints have to be reset so that the next call to the library function is detected as well. In a multi-threaded program, where several threads might be executing the same code, it's difficult to keep them from colliding with each other, and making them hit the breakpoints in the correct order.

The previous method is not thread-safe, so we defined and used another option, which is to emulate the `call` instruction. Two breakpoints remain enabled all the time. The first breakpoint is set before the call, while the second is set just on the return. When a thread first hits the entry breakpoint, it is stopped. The tracer then

978-0-7695-4159-4/10 $26.00 © 2010 IEEE
DOI 10.1109/NSS.2010.81

calculates the offset of the `call` instruction and proceeds to modify the thread's registers and stack appropriately. The thread is then allowed to continue, until it hits the exit breakpoint, and the result of the call is recorded. This way, we don't have to unset any breakpoints, and thus more than one thread can safely run the breakpointed code. To find the starting point of each function, we make use of backtraces, as well as the usage of tainted data.

A. Linux Instrumentation

We implemented a `ptrace` -based tracing mechanism. For each system call, the memory area used as source, and the memory area used as destination was recorded. Tainted data gets propagated to destinations, if a memory copy/transfer function is applied. We are able to trace programs that the regular GNU/Linux tracers (`strace` , `ltrace`) could not deal with because of the multi-threading problem. The interaction between thread creation and `ptrace` is not simple, and there are several corner cases in which the behavior of a newly created thread does not follow the `ptrace` specification.

We set two breakpoints in each call, one *entry* breakpoint and one *exit* breakpoint. This allows to read the arguments before the call, and get the return value after the call (as done with system calls). A software breakpoint is set by replacing the bytes at the break address with an `int 3` assembler instruction, which has opcode `0xcc` . Generating a software interrupt, which the OS converts to a trap signal to be delivered to the traced program. This signals get rerouted to the tracer, which interprets it as a breakpoint without reinjecting it to the traced program.

B. Windows tracing

For our Windows tracing, we planned to use PyDBG [13] which is a Python Windows debugger, a core component in the PaiMei reverse engineering framework[12]. It allows us to create our own callback functions, to define what should be executed when a debugging event occurs. This is done by setting software breakpoints where the first byte of the operation code of the normal instruction is replaced by the interruption instruction `CC`. When the CPU hits this interruption code it halts and the debugger catches the interruption. The original opcode is stored in a breakpoint list. During an interruption the address of the instruction is looked up in this breakpoint list and the original code is written back to the address. The instruction gets executed and after the execution, while using persistent breakpoints, the breakpoint gets restored. In [13] a Python script for locating and tracking predefined "dangerous" function calls is presented. While executing this script, we encountered from time to time a multi-thread error. From our understanding, the problem lies in the software breakpoints as already mentioned in beginning of this section. Therefore, we could not rely on PyDBG as it is not thread-safe and we have directly used the Windows Debugging API. To deal with the multithreading issue, we do not write back the original operation code to the address of the instruction. Instead, we leave the interruption code `CC` all the time and simulate the registry execution. Similarly to the Linux implementation, we had to parse the binary file format (Petal file format used by the linker and loaders on Windows architectures), we had to detect the imported dynamic loaded libraries (DLL) and exported functions. One of the main issues that we had to solve consisted in learning which library is loaded in the case of multiple same named files on the system.

III. Game Theory

Game Theory provides us with the necessary mathematical techniques for analyzing strategic situations. Such a situation is composed of two or more players, all of them have their own strategies and motivations. By playing a specified strategy a player receives a reward, depending on the strategy chosen by other players.

A. Fuzzing Game Models

The Nash Equilibrium [9] is a solution concept of Game Theory, specifying optimal strategic choices for all players by reason that none of the players has any motivation to diverge from the Nash equilibrium because one player can not gain greater payoffs by choosing another strategy when all the other players choose the strategies given by the profile. To calculate the Nash equilibrium we need N as a set of players, A_i as a finite strategy set, and R_i as a payoff function.

- N $\rightarrow$ set of n players
- $A_i \rightarrow$ finite strategy set ($a_i \epsilon A_i$)
- $R_i : A \rightarrow \mathbb{R}$ is a payoff function, where $A = A_1 x ... x A_n$

For our fuzzing model we have two different concepts as payoff function. We want to see how heterogenous our fuzzing framework is. Heterogeneity can be expressed by the entropy as it represents the distribution of the functions over all available functions called by injecting one message. If we assume to have a total of m backtraces, then each message can be represented as a vector where q_i is the number of different values that were exercised on the backtrace i.

We define the entropy of a message q_t to be: $H(q_t) = -\sum_{i=1}^{m} r_{t,i} log(r_{t,i})$ where: $r_{t,i} = \frac{q_{t,i}}{\sum_{i=1}^{m} q_{t,i}}$ In this case the index t counts the inputs, and the subscript i stands for the backtrace i.

In order to have an overview of how well a fuzzer works, we take the best performance of the entropy $max(H(q_t))$ and we make this value dependent on the number of messages n sent. We do this temporal normalization to account for the generated number of fuzz tests. Good fuzz strategies that perform well over time will have high performance values. However, a fuzzing strategy that generates some high values in the early phases, but fail to sustain the performance will be penalized. $\frac{max(H(q_t))}{\log t}, 2 \leq t \leq n$

We can use power as payoff function. The power defines the amount of functions called by one message, without considering if it is x-time the same function or 1-time x different functions. The instantaneous power of an input q is defined as: $Power(q_t) = \sqrt{\sum_{i=1}^{m} q_{t,i}^2}$ Again we maximize the power and make it dependent on the number of messages sent. $\frac{max(Power(q_t))}{\log t}, 2 \leq t \leq n$

After having defined the payoff functions, we can use the Nash equilibrium to calculate the optimal strategy. A Nash equilibrium can either have a pure strategy, which provides a complete definition of how a player will play a game, or a mixed strategy, that is a randomization over a set of pure strategies. A mixed strategy set for player i is the set of probability distributions over the action set A_i described by the simplex operator $\triangle$.

$$\triangle(A_i) = \{q_i : A_i \rightarrow [0,1] \mid \sum_{i=0} q_i(a_i) = 1\} \equiv Q_i$$

for mixed strategies: $Q = \prod_i Q_i$ and $q = (q_i, q_{-i}) \epsilon Q$ Expected payoffs to player i from strategy profile q in mixed strategies are: $\mathbb{E}_{a \sim q}[R_i(a)] = \sum_{a \epsilon A} q(a) R_i(a)$ where $q(a) = \prod_{j=1}^{N} q_j(a_j)$

The essential meaning of a mixed strategy is that randomized actions can achieve a better average payoff, and the equilibrium in mixed strategies is associated with the probability distribution over the set of actions, where this equilibrium can be achieved. In a mixed strategy, actions are performed randomly according to the probability distribution function.

In our model, there are two players: the first player is the fuzzing tool, the second player is Nature. We will describe several fuzzing strategies. Nature's action consists in selecting a given implementation. The strategy model is natural for the fuzzing tool, but a thorough discussion for Nature's choices is needed. In our work, we focus more on driving a fuzzing process. We assume that in most cases, we will not know the origin of the tested implementation, but that either this one has been built on top of existing ones (and thus share some common features) or that developers will reuse existing code for the device under test. On a more general level, this takes also into account that software writers will follow a similar programming paradigm (and make similar errors) when having to implement a given functionality. Our model is justified if we assume that developers will either perform fuzz testing on their software, or try to follow security guidelines and safe coding approaches. In most game theoretical models, the analysis is interested in capturing and assessing the equilibrium points and thus identify for each player, the best strategy to follow. We aim more at identifying the best strategy for the fuzzing tool and are mildly interested in assessing the game from a device perspective. The payoff is defined in terms of fuzzing entropy and respectively fuzzing power. That means, that we have two different zero sum games. In the first game, the fuzzing tool tries to maximize the average power, while in the second game, the objective consists in maximizing the entropy. We aim here at capturing both the fact that high entropies correspond to a large number of functions that are called as well as the fact that high power values reflect many test values used over the backtraces. We have identified several strategies that leverage the existing link between the tainted data graph and the parse tree of the input data. These strategies are based on applying mutations at individual subtrees of an initial parse tree. We have identified a set of different mutation actions, some are context dependent - they can be applied to specific node types. Some mutations are:

- Replacing a subtree for typical invalid bytes (e.g. `%x00, %x07,%x1F, %xFF`)
- Replacing a subtree for anomalous lengthly strings (e.g. `%s` hundreds of times)
- Regenerate the subtree using its syntax grammar but choosing random reduction at each time.
- Regenerate the subtree using a random rule chosen from the syntax grammar.

The strategies that can be applied by KiF are related to the selection of the node (in the parse tree) on which mutation is performed. Each node in the parse tree can be associated with a rank. This rank is simply the number of functions that are called having an argument tainted by this node. Fuzzing is done as follows:

∗ *Step 1*: An initial input data INPUT is generated.

∗ *Step 2*: A list of nodes NODELIST is constructed, with all nodes from the parse tree corresponding to input data INPUT.

∗ *Step3*: One node is selected from the list NODELIST. A mutation is applied to the message that contains this node. A

new input data NEWINPUT is obtained.

∗ *Step4*: The new input NEWINPUT is run against the application.

∗ *Step5*: This new input is parsed. The nodes from its corresponding parse tree are added to the list NODELIST.

∗ *Step6*: The list NODELIST is updated and the selected node is removed.

∗ *Step7*: GoTo Step3.

This generic fuzzing algorithm is highly dependent on the selection process done in STEP 3. We have extracted the following selection strategies.

∗ *Strategy I.a* **Greedy**: At each step, we select the node having the highest rank. A mutation is applied to that node.

∗ *Strategy I.b* **Greedy with memory**: At each time step, a mutation is applied to the node having the highest rank. This node is afterwards removed from the list NODELIST in order to avoid the pitfalls of a local maxima.

∗ *Strategy II.a* **Probabilistic**: A mutation is applied to a node that is selected probabilistically. The selection probability is proportional to the ranks.

∗ *Strategy II.b* **Probabilistic with memory**: A mutation is applied to a node that is selected probabilistically like in the previous case. The node is afterwards removed from the list NODELIST.

∗ *Strategy III* **Expert user** the second technique bases mutations on templates written by an expert user in which the fields to be modified are defined.

∗ *Strategy IV* **Exploiting function calls** : the last technique identifies all the function calls used for each of the fields of the message and generates different type of mutation depending on the nature of the function call.

IV. Experimental Results

For this work, we considered two different SIP-phone applications, Linphone and SJphone. All the tests were done under Ubuntu 9.04. Our fuzzing framework KiF[1] is open source and publicly available. KiF is a mutation based fuzzer implemented in Python. It is capable of automatically building a protocol fuzzer using the underlying ABNF grammar specifications and considering fuzzing operations to be defined on associated subtrees.

As already mentioned in Section III we have different strategies, and we consider to have two different players, on one side the fuzzing framework and on the other side the system. For every player and for every strategy we consider two different payoff concepts, the entropy and the power. In this section, we will show the experimental values and the results for calculating the Nash equilibrium. We calculate the Nash equilibrium for two different game models - having the entropy as payoff function, having the power as payoff function. In both cases have a zero-sum game. The Nash equilibrium is calculated with the help of the software Gambit [7]. The payoff for a strategy and a set of messages is computed by using the payoff for the last message. According to the definition of the payoff, this latter takes into account the maximum value discounted over the number of messages.

[1]http://kif.gforge.inria.fr/

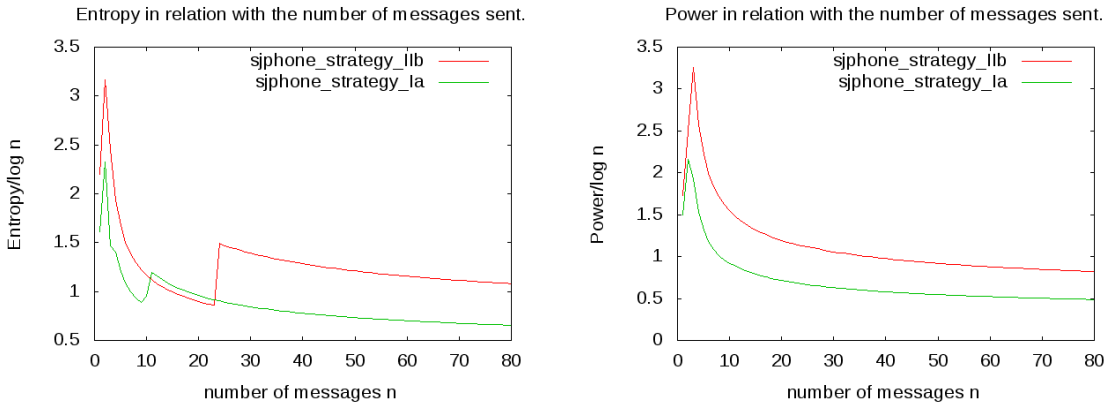
A. Entropy

In Figure 1(a) the behavior of two different payoff functions is presented, it shows that strategy II.b is underperforming over the first 22 input items, but manages to trigger an input item that significantly drives the payoff in higher value ranges. In Table I (a) lists the payoffs for the different strategies. Figure 2(a) shows the probability distribution for the entropy for the strategy I.b and respectively II.b. Strategy II.b is generating more input items having larger entropy values. The resulting Nash equilibrium consists of a pure strategy, where the optimal strategy II.b should be used for the fuzzing framework. In terms of implementation, SJphone performs best.

B. Power

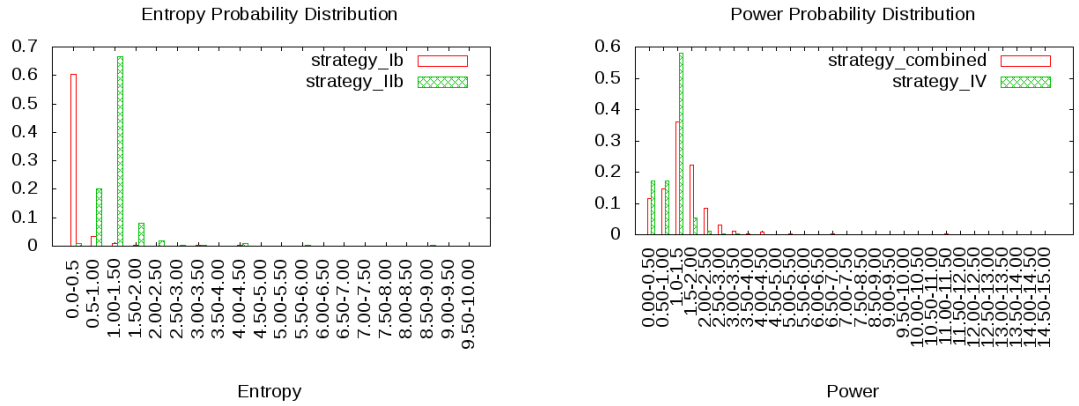
Figure 1(b) shows a payoff function behavior, where one strategy uniformly outperforms the other one. When considering the power as a payoff function, the results are summarized in Table I (b). The resulting Nash equilibrium consists of a mixed strategy, where the strategy II.b should be chosen with a probability of 0.1 and the strategy IV should be selected with a probability of 0.9. Figure 2(b) shows the probability distribution for the power for the strategy IV and the distribution for a combined strategy following the resulting mixed Nash equilibrium. The combined strategy uses the strategy IV with a probability of 0.9 and the strategy II.b with a probability 0.1. We have shown in that graph both distributions in order to highlight the advantage brought by combining both strategies. For instance, large power values are obtained by the mixed strategy. In tab:payoff(c) the resulting payoff functions are shown.

A natural question is related to the interest of having two different game models. While considering a payoff function that leverages the entropy, we look for strategies that explore different backtraces, it misses to capture the different values that are tested on each backtrace. Using a payoff that is based on the power, will allow to capture the different values that a backtrace is tested for. We have considered a global payoff function computed by the addition of the two normalized power and entropy payoffs, but we think that having two different game



(a) maximized Entropy over log n (b) maximized Power over log n

Fig. 1. Payoff functions



(a) Entropy distribution (b) Power distribution

Fig. 2. Probability distribution

models allows a finer grained control over the fuzz testing. From a practical perspective, a fuzz test should be performed in two major phase. In the first phase, the fuzz tool should aim at achieving a higher entropy and thus play the entropy game. At the end of this phase, a comprehensive set of backtraces should be obtained. In a follow-up phase, the power game should be played. In this game, the fuzz test will aim at maximizing the number of tests performed for each backtrace.

V. Related Work

A lot of work has already been done in the field of fuzzing. Hence, it is clear that many different approaches for this topic exist. One branch of work for fuzzing consists of using symbolic execution [4], [8], where input variables are made symbolic, and a set of constraints on these variables are assembled along an execution path. After that, a constraint solver generates test inputs satisfying the symbolic constraints. Even though, symbolic execution seems to be very effective, it also provides some drawbacks. The main drawback is that the source code has to be known. Another inconvenience is the execution duration. For source code of restricted size execution time is suitable, but for large embedded systems symbolic execution would consume to much time or even never finish. Another approach for fuzzing is tracing tainted data [14], [10], to follow the behavior of an application. A big advantage of taint tracing in comparison to symbolic execution is that we do not need to have knowledge of the source code, only the binaries are needed.

In this work, we use game theory to get the optimal fuzzing strategies. Previous work for applying game theory to a testing framework is described in [6] and in [3], where testing is considered as a game, in which the tester plays against the implementation under test. Two states,

TABLE I
PAYOFF FUNCTIONS

(a) Entropy

	Linphone	SJphone
Strategy I.a	1,23	0,76
Strategy I.b	0,98	0,82
Strategy II.a	1,33	0,74
Strategy II.b	1,07	0,84
Strategy III	0,94	0,72
Strategy IV	0,82	0,69

(b) Power

	Linphone	SJphone
Strategy I.a	1,85	1,10
Strategy I.b	2,25	0,69
Strategy II.a	2,03	0,86
Strategy II.b	2,56	0,72
Strategy III	1,26	1,47
Strategy IV	1,41	1,01

(c) Combined

	Linphone	SJphone
Strategy Ia	1,55	0,79
Strategy Ib	1,57	0,61
Strategy IIa	1,72	0,65
Strategy IIb	1,81	0,65
Strategy III	1,02	0,97
Strategy IV	1,00	0,69

active and passive, as well as two methods, controllable and observable actions, are assessed. The tester has a model for the behavior of the implementation under test. Reachability games are defined, where optimal strategies minimize the cost, knowing that every action is associated to a cost. Our previous work consisting fuzzing is presented in [1]. The first work where we approached a problem using a game-theoretical framework is described in [2].

VI. CONCLUSION & FUTURE WORK

In this paper, we presented a game theoretical model for analyzing different fuzzing mechanisms. Therefore, we assessed the efficiency of different fuzzing strategies with respect to the impact, based on the own-build instrumentation framework for tracing tainted data on a target application. The tracing is done using a tree-based representation of the parsing of tainted data, as well as the relevant code flow graph. For quantifying the impact we utilize the entropy metric, indicating how many different backtraces are tested, and the power metric, reflecting the different tested values for one or several input data items. These metrics are used for computing the payoff functions in our game theoretical model for the computation of the best fuzzing strategies. We deployed different fuzzing strategies against target applications. Following our experimental results, we deduce that the best strategies for fuzzing are a probabilistic mutation with memory and exploiting function calls.

In future work we will extend the current game to several other strategies and implementations. We plan to define payoff functions that take into account the underlying probabilistic distribution of vulnerabilities. For this purpose, we will need to make the link between the exploration of backtraces and the probability to discover a new vulnerability. Related to this work, we also plan to take into consideration the case of repeated games.

REFERENCES

[1] H. J. Abdelnur, R. State, and O. Festor. KiF: a stateful SIP fuzzer. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 47–56, New York, USA, 2007. ACM.

[2] Sheila Becker, Radu State, and Thomas Engel. Using game theory to configure p2p sip. In *IPTComm '09: Proceedings of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 1–9, New York, NY, USA, 2009. ACM.

[3] Andreas Blass, Yuri Gurevich, Lev Nachmanson, and Margus Veanes. Play to test. In *FATES*, pages 32–46, 2005.

[4] Cristian Cadar, Paul Twohey, Vijay Ganesh, and Dawson Engler. EXE: Automatically Generating Inputs of Death Using Symbolic Execution. In *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Virginia, USA, November 2006.

[5] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[6] Luca de Alfaro. Game models for open systems. In *Verification: Theory and Practice*, pages 269–289, 2003.

[7] Gambit, software tools for Game Theory. [online] gambit. sourceforge.net.

[8] P. Godefroid, A. Kieźun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *PLDI'2008: ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, US, 2008.

[9] Amy Greenwald. Matrix games and nash equilibrium. 2007.

[10] James Newsome, David Brumley, and Dawn Xiaodong Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.

[11] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[12] pedram.amini. PaiMei - reverse engineering framework. [online] http://code.google.com/p/paimei/.

[13] Justin Seitz. *Gray Hat Python: Python Programming for Hackers and Reverse Engineers*. No Starch Press, San Francisco, CA, USA, 2009.

[14] Martin Vuagnoux. Autodafé: an Act of Software Torture. In *Proceedings of the 22th Chaos Communication Congress*, pages 47–58, Berlin, 2005. Chaos Computer Club.