POWVER

Technical Report 2016-05

Title: **Facets of Software Doping**

Authors: Gilles Barthe, Pedro R. D'Argenio, Bernd Finkbeiner, Holger Hermanns

Report Number: 2016-05

ERC Project: Power to the People. Verified.

ERC Project ID: 695614

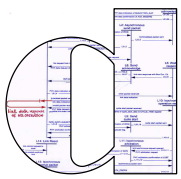
Funded Under: H2020-EU.1.1. – EXCELLENT SCIENCE

Host Institution: Universität des Saarlandes, Dependable Systems and Software

Published In: ISOLA (2) 2016

Please cite this publication as follows:

Gilles Barthe, Pedro R. D'Argenio, Bernd Finkbeiner, Holger Hermanns.
Facets of Software Doping.
Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II. Lecture Notes in Computer Science 9953, 2016, ISBN 978-3-319-47168-6. 601-608.



POWER TO THE PEOPLE.
VERIFIED.
erc

Facets of Software Doping*

Gilles Barthe[1], Pedro R. D'Argenio[2],
Bernd Finkbeiner[3], and Holger Hermanns[3]

[1] IMDEA Software
[2] FaMAF, Universidad Nacional de Córdoba – CONICET
[3] Saarland University – Computer Science, Saarland Informatics Campus

Abstract. This paper provides an informal discussion of the formal aspects of software doping.

1 Introduction

Software is the great innovation enabler of our times. Software runs on hardware. Usually, software is licensed to the hardware owner, instead of being owned by her. And while the owner is in full physical control of the hardware, she usually has neither physical nor logical control over the software. That software however does not always exploit the offered functionality of the hardware in the best interest of the owner. Instead it may be tweaked in various manners, driven by interests different from those of the owner or of society. This situation may be aggravated if the software is not running on local hardware but remotely (e.g. in the cloud) since the software user has now little or no control of its execution.

There is a manifold of facets to this phenomenon, summarised as software doping. It becomes more widespread as software is embedded in ever more devices of daily use. Yet, we are not aware of any systematic investigation or formalisation from the software engineering perspective.

This paper reviews known real cases of software doping, and provides a conceptual account of characteristic behaviour that distinguishes doped from clean software.

2 Software Doping in the Wild

The simplest and likely most common example of software doping is that of ink printers [4] refusing to work when supplied with a toner or ink cartridge of a third party manufacturer [8], albeit being technically compatible. More subtle variations of this kind of doping just issue a warning message about the risk of using a "foreign" cartridge [11]. In the same vein, it is known that printers

* This work is partly supported by the ERC Grants 683300 (OSARES) and 695614 (POWVER), by the Sino-German CDZ project 1023 (CAP), by ANPCyT PICT-2012-1823, by SeCyT-UNC 05/BP12 and 05/B497, and by the Madrid Region project S2013/ICE-2731 N-GREENS Software-CM.

emit "low toner" warnings [12] earlier than needed, so as to drive or force the customer into replacing cartridges prematurely. Similarly, cases are known where laptops refuse to charge [3] the battery if connected to a third-party charger.

Characteristic for these examples is that the functionality in question is in the interest of the device manufacturer, but against the customer interest. However, there are also variations of software doping that can be considered to be in the interest of the customer, but not in the interest of society: In the automotive sector, "chip-tuning" [19] is a remarkable variation of the software doping phenomenon, where the owner initiates a reprogramming of some of the vehicle's electronic control units (ECU) so as to change the vehicle characteristics with respect to power, emissions, or fuel consumption. By its nature, chip-tuning appears to be in the owner's interest, but it may well be against the interest of society, for instance if legally-defined and thus built-in speed limitations are overridden. Examples include scooters [9] and electric bikes [2] [10].

Some cases of software doping are clearly neither in the interest of the customer, nor in the interest of society. This includes as prominent examples the exhaust emission scandal of Volkswagen [20] (and other manufacturers). Here, the exhaust software was manufactured in such a way that it heavily polluted the environment, unless the software detected the car to be fixed on the particular test setup used to determine the NOx footprint data officially published.

The same sort of behaviour has been reported in the context of smart phone designs [7], where software was tailored to perform better when detecting it was running a certain benchmark, and otherwise running in lower clock speed. Another smart phone case, disabling the phone [5] via a software update after "non-authorized" repair, has later been undone [1]. Often, software doping is a part of a lock-in strategy: The customer gets *locked-in* on the manufacturer or unit-supplier for products, maintenance and services [13].

3 Characterising Software Doping

It is difficult to come up with a crisp characterisation of what constitutes software doping. Nevertheless we consider it a worthwhile undertaking to explore this issue, with the intention to eventually enable a formal characterisation of software doping. That characterisation can be the nucleus for formulating and enforcing rigid requirements on embedded software driven by public interest, so as to effectively ban software doping. In order to sharpen our intuition, we offer the following initial characterisation attempt.

> A software system is doped if the manufacturer has included a hidden functionality in such a way that the resulting behaviour intentionally favors a designated party, against the interest of society or of the software licensee. (1)

So, a doped software induces behaviour that can not be justified by the interest of the licensee or of society, but instead serves another usually hidden

interest. It thereby favors a certain brand, vendor, manufacturer, or other market participant. This happens intentionally, and not by accident.

However, the question whether a certain behaviour is intentional or not is very difficult to decide. To illustrate this, we recall that the above mentioned iPhone-6 case, where "non-authorized" repair rendered the phone unusable [5] after an iOS update, seemed to be intentional when it surfaced, but was actually tracked down to a software glitch of the update and fixed later. Notably, if the iOS designers would have had the particular intention to mistreat licensees who went elsewhere for repair, the same behaviour could well have qualified as software doping in the above sense (1).

As a result, we will look at software doping according to the above characterisation, but without any attempt to take into account considerations of intentionality.

In the sequel, we shall investigate this phenomenon by synthetic examples, that however are directly inspired by the real cases of software doping reviewed above.

3.1 Doping by discrimination

Think of a program as a function that accepts some initial parameters and, given (partial) inputs, it produces (partial) outputs. As an example, (an abstraction of) the embedded software in a printer is given in Fig. 1. The program PRINTER has the parameter *cartridge_info* (which is not yet used within the function), two input variables (*document* and *paper_available?*) and two output variables (*alert_signal* and *page_out*).

```
procedure PRINTER(cartridge_info)
    READ(document)
    while PAGESTOPRINT(document) > 0 do
        READ(paper_available?)
        if ¬paper_available? then
            TURNON(alert_signal)
            WAITUNTIL(paper_available?)
            TURNOFF(alert_signal)
        end if
        PRINTNEXTPAGE(page_out,document)
    end while
end procedure
```

Fig. 1. A simple printer.

A printer manufacturer may manipulate this program in order to favor its own cartridge brand. An obvious way is displayed in Fig. 2. This is a sort of discrimination based on parameter values. Therefore, a first formal approach to characterising a program as *clean* (or *doping-free*) is that it should behave in a similar way for all parameter values, where by *similar behaviour* we mean that the visible output should be the same for any

```
procedure PRINTER(cartridge_info)
    if BRAND(cartridge_info) = my-brand then
        (··· same code as Fig. 1 ···)
    else
        TURNON(alert_signal)
    end if
end procedure
```

Fig. 2. A doped printer.

```
procedure PRINTER(cartridge_info)
    READ(document)
    if ¬NEWTYPE(document) ∨ SUPPORTSNEWTYPE(cartridge_info) then
        (··· proceed to print as in Fig. 1 ···)
    else
        TURNON(alert_signal)
    end if
end procedure
```

Fig. 3. A clean printer.

given input in two different instances of the same (parameterized) program. Obviously, "all parameter values" refers to all values within a given domain. In the case of the printer, we expect that it works with any *compatible* cartridge. Such compatibility domain defines a first scope within which a software is evaluated to be clean or doped. So, we could say the following.

> A program is *clean* (or *doping-free*) if for every standard parameter it exhibits the same visible outputs when supplied with the same inputs. (2)

Under this view, the program of Fig. 2 is indeed doped. Also, note that this characterisation entails the existence of a contract which defines the set of standard parameters.

3.2 Doping vs. extended functionality

We could imagine, nonetheless, that the printer manufacturer may like to provide extra functionalities for its own product which is outside of the standard for compatibility. For instance (and for the sake of this discussion) suppose the printer manufacturer develops a new file format that is more efficient at the time of printing, but this requires some new technology on the cartridge. The manufacturer still wants to provide the usual functionality for standard file formats that works with standard compatible cartridges and comes up with the program of Fig. 3. Notice that this program does not conform to the specification of a clean program given by (2) since it behaves differently when a document of the new (non-standard) type is given. This is clearly not in the spirit of the program in Fig. 3 which is actually conforming to the standard specification. Thus, we relax the previous characterisation and only require that two instances of the program behave similarly if the provided inputs adhere to some expected standard. Therefore we propose the following weaker notion of clean program:

> A program is *clean* if for every standard parameter it exhibits the same visible outputs when supplied with any possible input complying with a given standard. (3)

This characterisation is based on a comparison of the behaviour of two instances of a program, each of them responding to different parameter values.

A second, different characterisation may instead require to compare a reference specification capturing the essence of clean behaviour against any possible instance of the program. The first approach seems more general than the second one in the sense that the specification could be considered as one of the possible instances of the (parameterized) program. However, the second characterisation is still reasonable and it could turn to be equivalent to (3) under mild conditions (namely, under behavioural equivalence.)

3.3 Doping by switching

Let us draw the reader's attention to a different facet of software doping. We consider the ECU of a diesel vehicle, in particular its exhaust emission control module. For diesel engines, the controller injects a certain amount of a specific fluid (an aqueous urea solution) into the exhaust pipeline in order to lower NOx emissions. We simplify this control problem to a minimal toy example. In Fig. 4 we display a function that reads the *throttle* position and calculates which is the dose of diesel exhaust fluid (DEF) that should be injected to reduce the NOx emission (this is stored in *def_dose*). Variable *throttle* is an input variable while, though *def_dose* is an output variable, it is not the actual visible output. The actual visible output is the NOx emission measured at the end of the exhaust system. Therefore, the behaviour of this system needs to be analyzed through testing. In this setting, we may only consider the standard input behaviour as the one defined in the laboratory emission tests.

```
procedure EmissionControl()
    read(throttle)
    def_dose = SCRModel(throttle)
end procedure
```

Fig. 4. A simple emission control.

The Volkswagen emission scandal arose precisely because their software was instrumented so that it works as expected *only if* operating in or very close to the lab testing conditions [6]. For our simplified example, this behaviour is exemplified by the algorithm of Fig. 5. Of course, the real case was less simplistic. Notably, a software like this one still meets the caracterization of *clean* given in (3). However, it is intentionally programmed to defy the regulations when being unobserved and hence it falls directly within our intuition of what a doped software is (see (1)).

```
procedure EmissionControl()
    read(throttle)
    if throttle ∈ throttleTestValues then
        def_dose = SCRModel(throttle)
    else
        def_dose = alternateSCRModel(throttle)
    end if
end procedure
```

Fig. 5. A doped emission control.

The spirit of the emission tests is to verify that the amount of NOx in the car exhaust gas does not exceed a given threshold *in general*. Thus, one would expect that if the input values of the EmissionControl function deviates within "reasonable distance" from the *standard* input values provided during the lab emission test, the amount of NOx found in the exhaust gas is still within the

regulated threshold, or at least it does not exceed it more than a "reasonable amount". Similar rationale could be applied for regulation of other systems such as speed limit controllers in scooters and electric bikes. Therefore, we propose this alternative characterisation:

> A program is *clean* if for every standard parameter, whenever it is supplied with any input (being it complying to the standard or not) that deviates within "reasonable distance" from a given standard input, it exhibits a visible output which does not deviate beyond a "reasonable distance" from the specified output corresponding to such standard input. (4)

The "reasonable distances" are values that should be provided (together with the notion of distance) and are part of the contract that ensures that the software is clean. Also, the limitation to this "reasonable distance" has to do with the fact that, beyond it, particular requirements (e.g. safety) may arise. For instance, a smart battery may decide to stop accepting charge if the current emitted by a standardized but foreign charger is higher than "reasonable", but it may still proceed in case it is instead dealing with a charger of the same brand for which it may know that it can resort to a customized protocol allowing ultra-fast charging in a safe manner.

These 'reasonable distances' need to come with application-specific metrics on possible input and output values. Since these metrics are often related to real physical quantities, the metric spaces might be continuous. They might also be discrete, or superpositions of both.

Characterisation (4) also plays a role when inputs and outputs cannot be precisely defined. This situation arises in cases like the exhaust emission system and almost any embedded system: input and output values will be as precise as sensors and actuators allow. In this case, the "reasonable distance" is going to be defined according to the precision of these devices.

4 Concluding remarks

This paper has reviewed facets of software doping. Starting off from real examples a first intuitive characterisation of software doping was derived. We then discussed a sequence of – still informal — definitions of absence of software doping.

We are currently working on the formalisation of these definitions. We expect that many definitions will fall in the general class of hyperproperties [15]—informally, hyperproperties are sets of sets of program executions and capture behaviours of multiple runs of a program—which encompasses continuity [14] and non-interference [16, 18]. These formal characterisations are expected to help to understand better the requirements on embedded software imposed by public interest, hence providing a framework to specify contracts or regulations pertaining to such technology, and to rigorously discriminate between doping and reasonably acceptable deviations from the normal behaviour. They will also

help clarify the specificities of software doping with respect to malware, software sabotage, and substitution attacks that have been studied in the context of security [17]. Furthermore, rigorous definitions will provide the necessary foundations for developing analysis methods (verification or testing) against doping.

References

1. Apple apologizes and updates iOS to restore iPhones disabled by error 53. https://techcrunch.com/2016/02/18/apple-apologizes-and-updates-ios-to-restore-iphones-disabled-by-error-53/, Online; accessed: 2016-07-07
2. BionX tuning. http://www.ebiketuning.com/comparison/bionx-tuning.html, Online; accessed: 2016-07-07
3. Dell laptops reject third-party batteries and AC adapters/chargers. Hardware vendor lock-in? https://nctritech.wordpress.com/2010/01/26/dell-laptops-reject-third-party-batteries-and-ac-adapterschargers-hardware-vendor-lock-in/, Online; accessed: 2016-07-07
4. Epson firmware update = no to compatibles. http://www.wasteink.co.uk/epson-firmware-update-compatible-problem/, Online; accessed: 2016-07-07
5. 'Error 53' fury mounts as Apple software update threatens to kill your iPhone 6. https://www.theguardian.com/money/2016/feb/05/error-53-apple-iphone-software-update-handset-worthless-third-party-repair, Online; accessed: 2016-07-07
6. The exhaust emissions scandal ("Dieselgate"). https://events.ccc.de/congress/2015/Fahrplan/events/7331.html, Online; accessed: 2016-07-07
7. Galaxy S4 on steroids: Samsung caught doping in benchmarks. http://forums.appleinsider.com/discussion/158782/galaxy-s-4-on-steroids-samsung-caught-doping-in-benchmarks, Online; accessed: 2016-07-07
8. Has a printer update rendered your cartridges redundant? https://conversation.which.co.uk/technology/printer-software-update-third-party-printer-ink/, Online; accessed: 2016-07-07
9. How it works: The basics of ECU tuning. https://rideapart.com/articles/hows-work-ecu-tuning, Online; accessed: 2016-07-07
10. How to get access to your bionx console code menu (codelist included). http://electricbikereview.com/community/threads/how-to-get-access-to-your-bionx-console-code-menu-codelist-included.519/, Online; accessed: 2016-07-07
11. The secret printer companies are keeping from you. http://uk.pcmag.com/printers/60628/opinion/the-secret-printer-companies-are-keeping-from-you, Online; accessed: 2016-07-07
12. Take that, stupid printer! http://www.slate.com/articles/technology/technology/2008/08/take_that_stupid_printer.html, Online; accessed: 2016-07-07
13. Arthur, W.B.: Competing technologies, increasing returns, and lock-in by historical events. The Economic Journal 99(394), pp. 116–131 (1989), http://www.jstor.org/stable/2234208

14. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity analysis of programs. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'10. pp. 57–70 (2010)
15. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of CSF'08. pp. 51–65 (2008)
16. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982. pp. 11–20. IEEE Computer Society (1982), http://dx.doi.org/10.1109/SP.1982.10014
17. Schneier, B., Fredrikson, M., Kohno, T., Ristenpart, T.: Surreptitiously weakening cryptographic systems. Cryptology ePrint Archive, Report 2015/097 (2015), http://eprint.iacr.org/2015/097
18. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security 4(2/3), 167–188 (1996), http://dx.doi.org/10.3233/JCS-1996-42-304
19. Wikipedia: Chip tuning —Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Chip_tuning (2016), Online; accessed: 2016-07-07
20. Wikipedia: Volkswagen emissions scandal — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Volkswagen_emissions_scandal (2016), Online; accessed: 2016-07-07