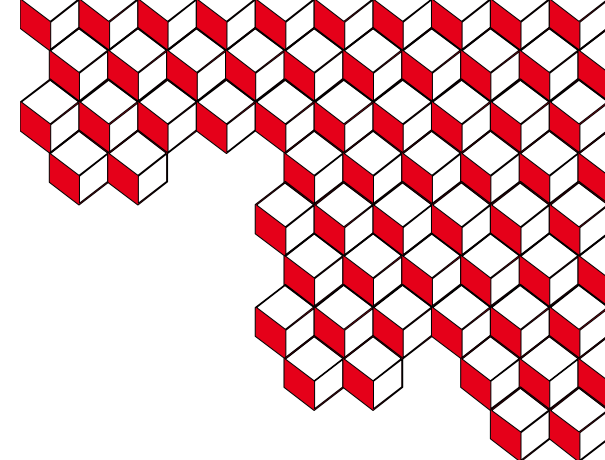




list

université
PARIS-SACLAY

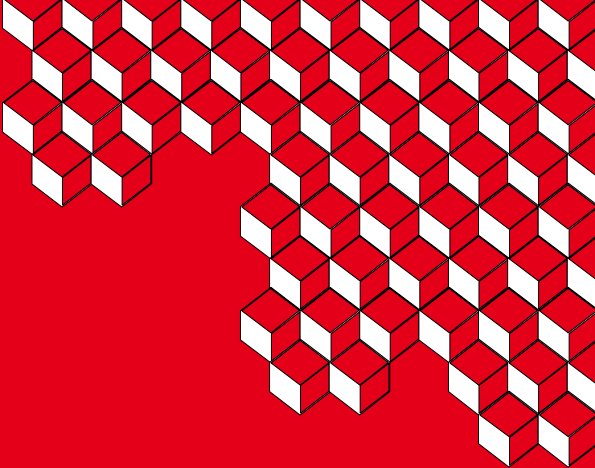


Triggering the Hidden: Oracle Design Meets KLEE for Backdoor Detection

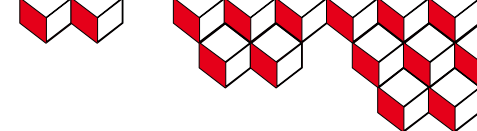
Michaël Marcozzi *(includes work done with Dimitri Kokkonis and Stefano Zacchirolì)*

Munich, October 16, 2026

Fifth KLEE workshop



“ *Why this talk?* ”



Several months ago, near Paris...

What do
you do?



Cristian

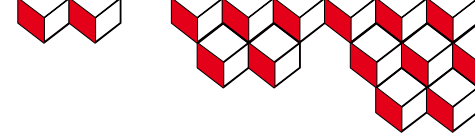


Dimitri

I do a PhD on
backdoor detection
with fuzzing!



Michaël



ROSA: Finding Backdoors with Fuzzing

Dimitri Kokkonis, Michaël Marcozzi, Emilien Decoux
Université Paris-Saclay, CEA, List
Paris-Saclay, France
first.last@cea.fr

Stefano Zacchiroli
LTCL, Télécom Paris, Institut Polytechnique de Paris
Palaiseau, France
stefano.zacchiroli@telecom-paris.fr

Abstract—A code-level backdoor is a hidden access, programmed and concealed within the code of a program. For instance, hard-coded credentials planted in the code of a file server application would enable maliciously logging into all deployed instances of this application. Confirmed software supply-chain attacks have led to the injection of backdoors into popular open-source projects, and backdoors have been discovered in various router firmware. Manual code auditing for backdoors is challenging and existing semi-automated approaches can handle only a limited scope of programs and backdoors, while requiring manual reverse-engineering of the audited (binary) program. Graybox fuzzing (automated semi-randomized testing) has grown in popularity due to its success in discovering vulnerabilities and hence stands as a strong candidate for improved backdoor detection. However, current fuzzing knowledge does not offer any means to detect the triggering of a backdoor at runtime.

In this work we introduce ROSA, a novel approach (and tool) which combines a state-of-the-art fuzzer (AFL++) with a new metamorphic test oracle, capable of detecting runtime backdoor triggers. To facilitate the evaluation of ROSA, we have created ROSARUM, the first openly available benchmark for assessing the detection of various backdoors in diverse programs. Experimental evaluation shows that ROSA has a level of robustness, speed and automation similar to classical fuzzing. It finds all 17 authentic or synthetic backdoors from ROSARUM in 1h30 on average. Compared to existing detection tools, it can handle a diversity of backdoors and programs and it does not rely on manual reverse-engineering of the fuzzed binary code.

Index Terms—fuzzing, dynamic analysis, metamorphic testing, backdoors, vulnerability detection

I. INTRODUCTION

Context. A code-level backdoor [1] is a hidden access, programmed and concealed within the code of a program. It enables program users aware of the backdoor to feed the program with a specific input value and thus trigger a privilege escalation within the program or gain undue access to underlying system resources. For example, hard-coded credentials planted in the code base of a file server application can enable maliciously logging into all deployed instances of this application in the world. Confirmed software supply-chain attacks have led to the injection of backdoors into popular open-source projects, like PHP [2], ProFTPD [3], vsFTPd [4] and xz [5]. Backdoors have also been discovered in the binary firmware of popular network routers [6]–[9].

Graybox fuzzing [10] is a form of automated program testing. It relies on a search-based approach [11] to generate test inputs automatically and on simple test oracles [12]

(such as crash detection and code sanitizers [13]) to detect failures automatically at runtime. Among advanced capabilities, modern fuzzing tools (or *fuzzers*), like the community-maintained AFL++ [14], are now equipped for testing binary-only programs [15] and for efficiently exploring complex branching conditions in the tested code [16], [17]. These tools are currently attracting a lot of popularity and research efforts, notably because of their reported ability [14] to discover software vulnerabilities in programs.

Problem. Addressing the threat of backdoors requires proper auditing of software dependencies and (binary) firmware. Yet, this necessitates a long and painstaking manual inspection of large amounts of code, so that auditing is often not performed at all [18]. While there has been some progress in automating backdoor detection [3], [18]–[20], the state of the art still suffers from the limited scope of programs and backdoors that can be handled. In addition, current detection tools still rely on manual reverse-engineering of the vetted (binary) code.

Goal and challenges. In this work, we aim at taking advantage of the capabilities of modern graybox fuzzers to automate backdoor detection. Fuzzing has indeed the potential to enable backdoor detection for a wide variety of programs and backdoors, with no manual code reverse-engineering. Yet, while the current body of knowledge in fuzzing enables generating test inputs for a wide range of programs, it does not offer any means to detect the triggering of a backdoor at runtime. In addition, benchmarking backdoor detection capabilities over multiple programs and backdoors is difficult, as backdoor reports in the literature are scarce and often point to lost samples or undocumented binary firmware, running on obsolete and difficult-to-obtain appliances.

Proposal. We introduce ROSA, a novel approach (and tool) which combines a state-of-the-art fuzzer (AFL++) with a new metamorphic test oracle [21], capable of detecting backdoor triggers at runtime. The key intuition behind ROSA is that, for example, fuzzing a backdoored file server application with incorrect credentials should always cause *similar observable reactions*; however, among the generated wrong credentials, the ones that trigger the backdoor will cause a *different reaction*, enabling the ROSA oracle to detect them.

To facilitate the experimental evaluation of ROSA and its comparison with existing tools, we have created and made available the novel ROSARUM benchmark, consisting of 7



Not In My Git Yard: Catching Backdoors at Commit and Release Time

Dimitri Kokkonis^{*}
dimitri.kokkonis@cea.fr
Université Paris-Saclay
CEA, List
Paris-Saclay, France

Michaël Marcozzi
michael.marcozzi@cea.fr
Université Paris-Saclay
CEA, List
Paris-Saclay, France

Stefano Zacchiroli
stefano.zacchiroli@telecom-paris.fr
LTCL, Télécom Paris
Institut Polytechnique de Paris
Palaiseau, France

Abstract

Code-level backdoors—stealthy code changes that grant hidden privileges via secret triggers—pose a persistent threat to open-source software. Known attempts to inject such backdoors into widely used projects through malicious commits, tampered release packages, or compromised third-party dependencies, were stopped only by luck and manual review. Existing Continuous Integration (CI) pipelines cannot detect these attacks, and downstream binary analysis tools require substantial manual effort. In this work, we present LIT, an automated approach that strengthens open-source development and release processes against backdoor injection. LIT integrates a backdoor detection mechanism into (1) CI pipelines to block malicious commits, and (2) release vetting workflows to prevent tampered releases or compromised dependencies from entering large ecosystems, such as Linux distributions. LIT offers two key contributions. First, it enhances CI-compatible fuzzing with the capability to detect triggers of suspicious behavior based on historical and current software executions. This enables fast, precise backdoor detection suitable for both CI and update validation workflows. Second, it combines code change analysis with fuzzing data to precisely point maintainers to backdoor-revealing code regions, even when release updates modify millions of lines of code. We also outline five strategies attackers could use to evade LIT, and evaluate corresponding defenses. Our experiments across hundreds of benign and backdoored commits and releases show that LIT achieves high detection accuracy with low false alarm rates, reliably identifies malicious code, resists adversarial attempts, and would have prevented real-world backdoor incidents.

CCS Concepts

• Security and privacy → Software security engineering: Malware and its mitigation; • Software and its engineering → Software testing and debugging.

Keywords

Fuzzing, Dynamic Analysis, Backdoors, Continuous Integration, Release Vetting, Software Supply Chain Security



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2842-2/2026/18
<https://doi.org/10.1145/3832783.3834352>

ACM Reference Format:

Dimitri Kokkonis, Michaël Marcozzi, and Stefano Zacchiroli. 2026. Not In My Git Yard: Catching Backdoors at Commit and Release Time. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3832783.3834352>

1 Introduction

Context. Code-level backdoors [21, 45]—hidden functionalities embedded within source code that allow individuals aware of their presence to obtain elevated privileges or unauthorized features via secret triggers—remain a persistent threat to open-source software. Incidents involving malicious commits to the PHP public repository [1], tampering with ProFTPD and vsFTPd release packages [28, 29], or the injection of backdoors through a compromised dependency to XZ Utils [30] have gone undetected for days, being uncovered only through a combination of manual inspection and chance. The consequences of missing such threats are severe, especially as the ever-expanding ecosystem of software dependencies magnifies the potential impact of a single compromise [7].

Problem. The current open-source software development and release stack lacks systematic defenses against backdoor injection attacks. Upstream, *Continuous Integration (CI) pipelines* run automated tests—typically on every commit—to ensure code quality through compilation checks and regression testing. *Fuzzing*, already integrated into CI pipelines of major projects [31, 40–42], targets crashes and memory safety bugs through brute-force testing of the revised code. Because CI must provide rapid feedback, fuzzing runs are brief, often limited to about 30 minutes [10, 17, 20]. Existing CI mechanisms, however, cannot detect backdoored commits such as hard-coded credentials or hidden reverse shells. A recent machine-learning approach that analyzes Git metadata [12] could offer a partial solution but still yields many false alarms. Downstream, state-of-the-art backdoor detection tools [21, 34, 37, 44, 46] help end users vet binaries, but demand manual effort to reverse-engineer code or to dismiss false alarms, limiting scalability.

Goal and Challenges. We aim to harden open-source development and release pipelines by introducing a mechanism that blocks most backdoor injection attacks, integrating it in two stages: (1) in CI pipelines, and (2) in release vetting before updated packages enter major ecosystems such as Linux distributions. Doing so requires addressing three challenges:

(1) CI and release vetting pipelines operate under strict time constraints, so backdoor detection must be fully automated and able to achieve a high detection rate within these limits to warrant

ICSE'25

ASE'26

Why not
use
symbolic
execution?



Fuzzing offers better trade-
offs to sweep across large
codebases in search for
backdoors...



Why not
use
symbolic
execution?

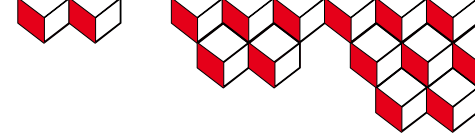


Fuzzing offers better trade-
offs to sweep across large
codebases in search for
backdoors...



But could not
symbolic execution
be useful here too?

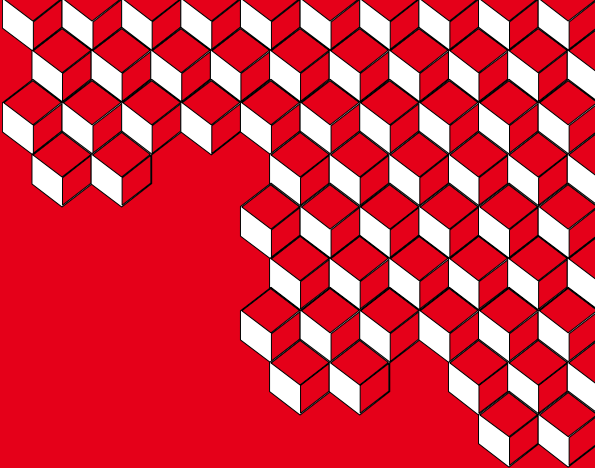




The rest of this talk...

Part 1: our work on backdoor detection with fuzzing

Part 2: my thoughts about how symbolic execution could help



“ *Backdoor detection with fuzzing* ”

ICSE'25 and ASE'26

joint work with Dimitri Kokkonis and Stefano Zacchiroli

Outline

1. Code-level backdoors
2. Detection
3. Our test oracle
4. Oracle-based fuzzing
5. Experimental evaluation

Outline

1. Code-level backdoors
2. Detection
3. Our test oracle
4. Oracle-based fuzzing
5. Experimental evaluation

Backdoors and code-level backdoors

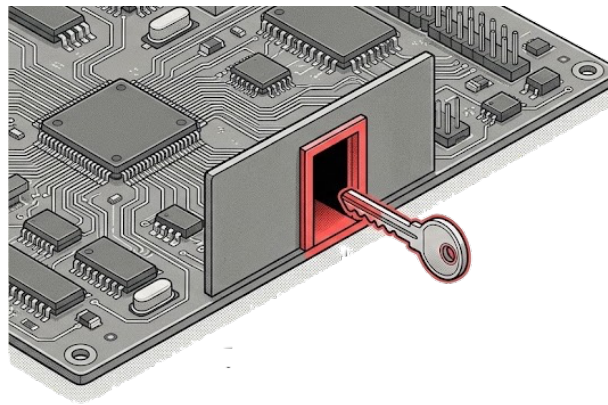
Backdoor = hidden access to a system, bypassing known control points

Various backdoors found in whole IT stack:

- Backdoors in *hardware, cryptographic algorithms, configurations...*
- *Software backdoors*: neural network poisoning, bugdoors, **code-level backdoors...**

Code-level backdoor = explicit **feature in the code** of traditional software:

- Triggered by attacker-known *atypical inputs*
- Enabling *privilege escalation* or *unauthorized features* (e.g. login bypass, reverse shell)



Injecting code-level backdoors

Known code-level backdoors were inserted by:

Compromising open-source projects (malicious pull requests, maintainer infiltration...)

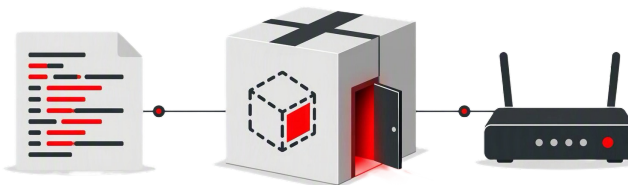
- *XZ Utils* (CVE-2024-3094) and *PHP git repository attack* (2021)

Tampering with software distribution (hacked package download websites...)

- *ProFTPD* and *vsFTPd* backdoors (CVE-2010-20103 and CVE-2011-2523)

Shipping infected firmware binaries (reverse shells in IoT devices...)

- HTTP daemons on some *Belkin* or *Tenda* routers allow remote command execution



Example: the vsFTPD attack (CVE-2011-2523)

Attackers **gained access** to the **vsFTPD distribution server**

They replaced vsftpd-2.3.4.tar.gz with an **infected copy**

Users who downloaded it between June 30 and July 3, 2011 **found their machines backdoored:**

- Trigger: FTP login with a *username containing a smiley face* :)
- Result: a *remote* interactive command *shell* listens on port 6200



Outline

1. Code-level backdoors
- 2. Detection**
3. Our test oracle
4. Oracle-based fuzzing
5. Experimental evaluation

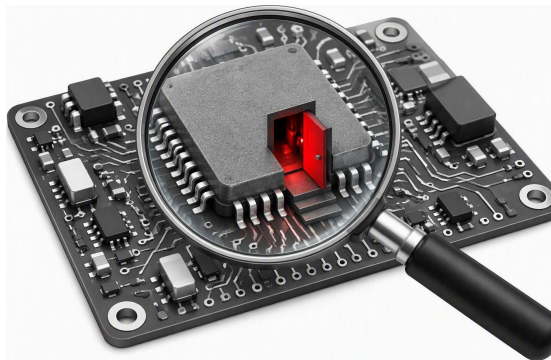
Related work: firmware auditing

Four decade-old **tools** support **manual firmware analysis to detect backdoors**:

- Weasel ^{CCS'13} --- *dynamically identifies code units prone to backdoors*
- Firmalice ^{NDSS'15} --- *analyzes paths to authenticated code using angr*
- Humidify ^{DIMVA'17} --- *uses ML to identify program types and compare feature profiles*
- Stringer ^{ESORICS'17} --- *statically identifies strings that strongly influence control flow*

Nice! But, limitations:

- *Extensive manual reverse engineering to define targets or review results*
- Only Stringer is *not deprecated*. It often *misses backdoors* and flags *hundreds of false positives*
- No standard detection *benchmark*. Most backdoor samples are *lost*



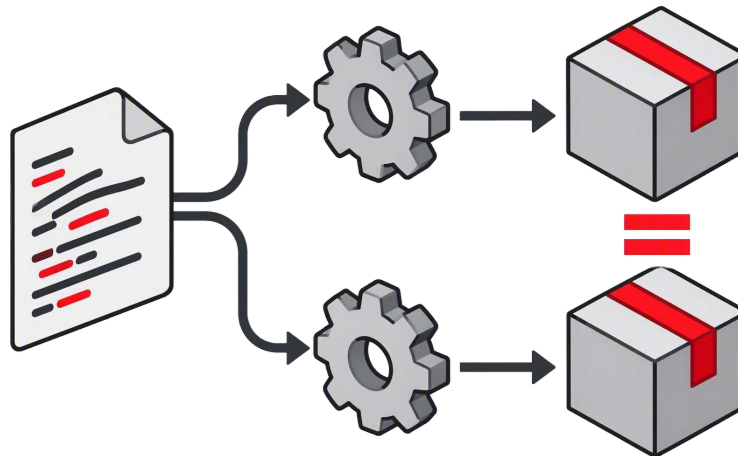
Related work: open-source software protection

Ganz et al. CODASPY'23 use **ML on metadata** to flag **suspicious Git contributions**

- **Limitations:** appears prone to frequent false positives; evasion risks remain unclear

Reproducible builds produce bit-for-bit identical binaries from the same source code

- This enables *independent verification of build integrity* (**but nothing more**)



To sum up

Code-level backdoors pose a **real, high-impact threat**

- E.g., the XZ Utils backdoor would have given a state actor SSH access to most servers worldwide

We need highly automated, effective and precise backdoor detection tools

- We also need an archived, standardized benchmark for backdoor detection



Outline

1. Code-level backdoors
2. Detection
3. Our test oracle
4. Oracle-based fuzzing
5. Experimental evaluation

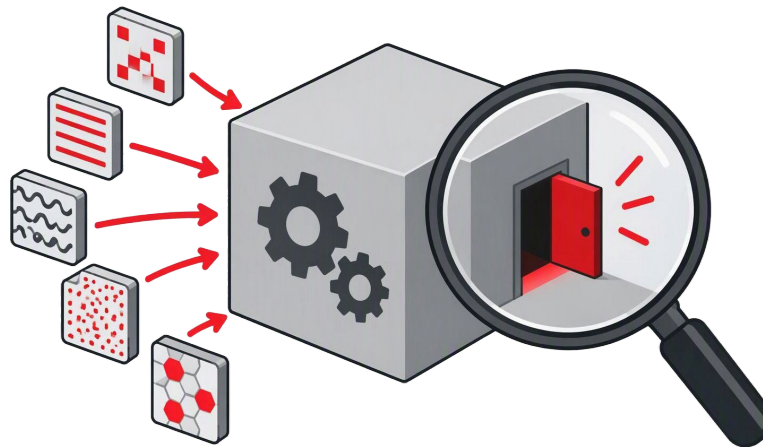
Detecting backdoors with fuzzing

Fuzzing with crash monitoring enabled **automated, effective and precise memory vulnerability detection**

We aim to **extend this success to backdoor detection**

Fuzzing generates tests, but **we need an oracle to detect backdoor activation** at runtime

Challenge: design a *universal* oracle for *diverse* backdoors with *program-specific semantics*



A test oracle for backdoor activation

Shared trait of code-level backdoors: rare inputs trigger atypical program behavior

Our **backdoor activation** oracle uses two steps:

- **Step 1:** build a database of *typical program behaviors*
 - Run an adequate *test suite* and record *system call profiles*
 - A profile is the *set of system call types* used per input (e.g., read, execve), a proxy for program behavior
- **Step 2:** fuzz for *broad code coverage* and flag *new unseen profiles* as possible backdoors



Outline

1. Code-level backdoors
2. Detection
3. Our test oracle
4. Oracle-based fuzzing
5. Experimental evaluation

Rosa+Lily: our backdoor detection fuzzers

Rosa audits *binary firmware*. **Lily** vets *CI changes*

Why 2 different tools?

- Both combine **AFL++** with **CMPLOG** and our oracle
- Lily uses **source code** and **history** to meet **CI needs**: **speed** and very **low false-alarm** rates
- Lily flags atypical behavior only **if it is absent from the pre-change code**



Outline

1. Code-level backdoors
2. Detection
3. Our test oracle
4. Oracle-based fuzzing
5. Experimental evaluation

Introducing the ROSARUM benchmark

Our backdoor detection benchmark combines real samples with realistic synthetic ones:

Program			Backdoor	
Name	Type	Binary size	Origin	Description
Authentic backdoors				
Belkin / httpd	Router HTTP server	2.6 MiB	Router manufacturer	HTTP request with secret URL value leads to web shell [6]
D-Link / thttpd	Router HTTP server	7.2 MiB		HTTP request with secret field value bypasses authentication [7]
Linksys / scfgmgr	Router TCP server	2.5 MiB		Packet with specific payload enables memory read/write [9]
Tenda / goahead	Router HTTP server	2.9 MiB		Packet with specific payload enables command execution [8]
PHP	HTTP server	80.6 MiB	Supply-chain attack	HTTP request with secret field value enables command execution [2]
ProFTPD	FTP server	3.3 MiB		Secret FTP command leads to root shell [3]
vsFTPD	FTP server	2.9 MiB		FTP usernames containing " :) " lead to root shell [4]
Synthetic backdoors				
sudo	Unix utility	8.4 MiB	Paper example	Hardcoded credentials (see Listing 1)
libpng	Image library	7.0 MiB	Manual injection in the MAGMA [22] fuzzing benchmark	Secret image metadata values enables command execution
libsndfile	Sound library	6.6 MiB		Secret sound file metadata value triggers home directory encryption
libtiff	Image library	10 MiB		Secret image metadata value enables command execution
libxml2	XML library	8.2 MiB		Secret XML node format enables command execution
Lua	Language interpreter	3.7 MiB		Specific string values in script enables reading from filesystem
OpenSSL / bignum	Crypto library	12.2 MiB		Secret bignum exponentiation string enables command execution
PHP / unserialize	Language interpreter	30.2 MiB		Specific string values in serialized object enables PHP code execution
Poppler	PDF renderer	39.4 MiB		Secret character in PDF comment enables command execution
SQLite3	Database system	6.4 MiB		Secret SQL keyword enables removal of home directory

Rosa/Lily detection and false positive rates

Rosa: on avg., all but one backdoor detected in **90 min**, **7 atypical behaviors** to vet for false positives

Backdoor	ROSA — (10 runs × 8 hours) / backdoor — 1 minute of fuzzing for phase 1							
	Failed runs	Robustness + speed			Automation level			
		Time to first backdoor input			Baseline Avg. seeds	Manually inspected inputs		
		Min.	Avg.	Max.		Min.	Avg.	Max.
Authentic backdoors								
Belkin / httpd	10 / 10	Timeout	Timeout	Timeout	2773	2	4	6
+ with specialized seeds*	3 / 10	17m40s	3h49m29s	Timeout	2781	4	5	7
D-Link / thttpd	0 / 10	2m07s	15m00s	43m42s	3648	7	9	12
Linksys / scfgmgr	0 / 10	1m05s	1m29s	1m55s	251	1	1	1
Tenda / goahead	0 / 10	1m28s	3m34s	8m10s	535	1	2	2
PHP	1 / 10	24m30s	2h03m44s	Timeout	11631	4	8	16
ProFTPD	4 / 10	4m03s	3h37m32s	Timeout	2995	5	8	11
vsFTPd	0 / 10	3m04s	5m41s	11m03s	1890	3	4	4
Synthetic backdoors								
sudo	0 / 10	5m47s	8m05s	11m46s	167	1	1	1
libpng	2 / 10	13m47s	2h24m46s	Timeout	4202	1	2	2
libsndfile	3 / 10	2h21m08s	5h04m46s	Timeout	10376	9	12	13
libtiff	0 / 10	5m08s	12m15s	25m10s	9566	1	3	5
libxml2	0 / 10	8m17s	27m14s	1h09m06s	12104	9	14	20
Lua	1 / 10	50m34s	4h07m41s	Timeout	6653	6	12	17
OpenSSL / bignum	0 / 10	9m53s	22m00s	39m52s	1441	1	1	2
PHP / unserialize	0 / 10	23m05s	1h04m39s	1h35m08s	6285	1	1	1
Poppler	0 / 10	11m28s	49m09s	1h33m02s	9544	5	6	8
SQLite3	0 / 10	33m17s	1h02m52s	2h42m42s	4705	20	26	31

Lily: **90%** backdoor detection in **10-minute CI runs**, with a **0.2% false-positive** rate on safe commits

“ *Improving backdoor
detection
with symbolic execution* ”

Reminder: how our approach works...

Our **backdoor detection** uses two steps:

- **Step 1:** build a database of *typical program behaviors*
 - Run an adequate *test suite* and record *system call profiles*
 - A profile is the *set of system call types* used per input (e.g., read, execve), a proxy for program behavior
- **Step 2:** fuzz for *broad code coverage* and flag *new unseen profiles* as possible backdoors



How symbolic execution could step in!

Our **backdoor detection** uses two steps:

- **Step 1:** build a database of *typical program behaviors*
 - Run an adequate *test suite* and record *system call profiles*
 - A profile is the *set of system call types* used per input (e.g., read, execve), a proxy for program behavior

--- Static analysis of system call sites can recover possible backdoor paths with unseen type combinations

--- Symbolic execution could check feasibility for some of these paths before optionally moving to Step 2
- **Step 2:** fuzz for *broad code coverage* and flag *new unseen profiles* as possible backdoors



Issues we are investigating...

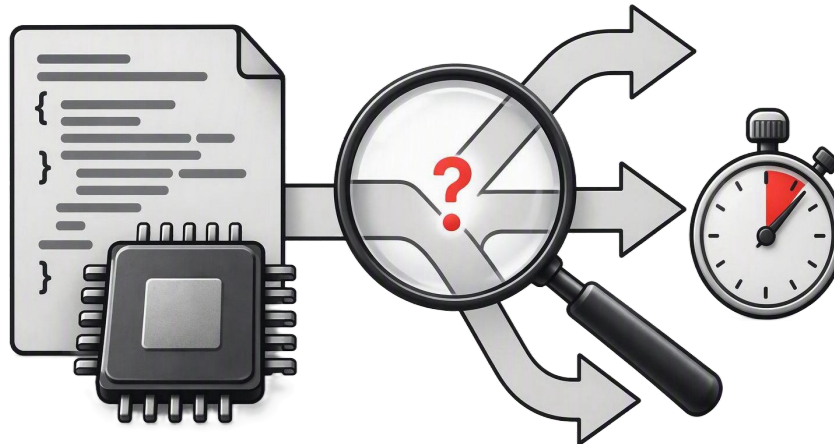
How can we **statically identify system call sites** in source and binary code?

How should we **prioritize promising paths** for symbolic execution?

How long should we analyse paths **before potentially switching to Step 2 fuzzing**?

Should we use **static or dynamic** symbolic execution?

How will the technique differ for **source-level CI analysis** and **binary-only firmware analysis**?



Takeaways

Code-level backdoors enter through open-source development, software distribution or infected firmware

We showed (ICSE'25, ASE'26) that **fuzzing with an oracle for atypical behavior** detects them efficiently

Symbolic execution could verify promising potential backdoor paths before fuzzing



Michaël Marcozzi
Tenured researcher

Université Paris-Saclay, CEA, List
France

<http://www.marcozzi.net>



I am looking for postdocs 🙋