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
LTCI, 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 backdooors 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

authentic backdoors, coupled with 10 diverse synthetic backdoors inserted into a standard fuzzing benchmark [22].

Evaluation. We run 10 fuzzing campaigns, lasting 8 hours each, using ROSA on each backdoor in the ROSARUM benchmark. ROSA can detect all 17 backdoors in 1h30 on average, demonstrating a level of robustness and speed similar to vanilla AFL++ for classical bugs. The automation level is also similar to AFL++, but ROSA may produce false positives that must then be semi-automatically discarded. Yet, the required manual effort is limited to vetting (an average of 7) suspicious runtime behaviors detected while fuzzing, like the launching of a root shell. This level of performance primarily qualifies ROSA as a good candidate to increase automation during large-scale code auditing events, like before deploying router firmware or software dependencies in critical infrastructures.

We compare in depth against STRINGER, the only competing backdoor detection tool that is available and working. As it relies on a simple static analysis, STRINGER is faster than ROSA, but can only detect 4 out the 17 ROSARUM backdoors and produces 44 times more false positives.

Contributions. Our main contributions are:

1) A new metamorphic oracle (based on a novel metamorphic relation and a fresh heuristic approach to find pairs of related inputs) which makes it possible, for the first time, to use graybox fuzzing as a means to detect code-level backdoors.

2) ROSA, an efficient backdoor detection method and tool, which complements a state-of-the-art fuzzer (AFL++) with our new metamorphic oracle. ROSA significantly improves the state of the art in backdoor detection, by (1) enabling the efficient discovery of a wider range of backdoors in a wider range of programs, compared to what existing approaches can do, and by (2) removing the need to manually reverse-engineer the analyzed (binary) code, which existing approaches still do require.

3) ROSARUM, the first openly available benchmark for evaluating backdoor detection tools, as well as largest backdoor dataset ever used as per the state of the art.

Data availability statement. ROSA and ROSARUM are available on GitHub and are archived on Software Heritage [23], [24]. A result replication package is available at https://zenodo.org/records/14724251.

II. BACKGROUND

A. Code-level backdoors

1) Definition and scope: A backdoor in a fortified castle is an unprotected but concealed access. It enables those who are informed of its existence to circumvent all castle fortifications and enter without effort. By analogy, a *backdoor in a computer system* is a hidden mechanism built by developers, able to grant undue privileges to users who are informed about it. Various instances of backdoors have been reported in diverse computer systems, like hardware backdoors in processors [25], mathematical backdoors in cryptographic algorithms [26] and data poisoning backdoors in machine learning models [27]. In

this work, we focus on *code-level backdoors* [1] in classical software programs, like file server applications or command-line tools. In a nutshell, code-level backdoors are hidden features programmed and concealed within the program code base. They enable informed program users to leverage a specific *key input value*, to trigger either a privilege escalation within the program or to get undue access to resources in the underlying system. For example, the key of a code-level backdoor can be a set of hard-coded credentials in the authentication module of a web server program [7] or a non-standard FTP command giving root access to a file server application's underlying operating system [3].

2) Occurrence in the real world: At least two types of software attacks in the real world have involved the secret injection of a code-level backdoor in a program. The first type is *software supply-chain attacks* [28], [29]. The vast majority of modern software relies on thousands of third-party open-source or proprietary dependencies [30], [31], and some of them come with backdoors. Injections of code-level backdoors in popular open-source software, such as PHP [2], ProFTPD [3], vsFTPd [4] and xz [5], have been reported. In addition, dubious software developers may sell proprietary components infected by deliberate or unintentional backdoors (like a forgotten debug access). The second identified type of attacks involving backdoors relates to *dubious Internet of Things (IoT) manufacturers*. IoT devices, like network or surveillance appliances, typically come with embedded proprietary software, called *firmware*, driving their operations. Injecting a code-level backdoor in such firmware can enable compromising millions of devices worldwide, in possibly critical infrastructures. Firmware infections with backdoors have been reported for various types of routers [6]–[9].

3) Detection: Addressing the threat of code-level backdoors requires a proper auditing of software dependencies and IoT firmware, to ensure that they do not contain any such backdoor. Yet, in practice, such auditing is difficult, so that it is either not done or it requires a long and painstaking code inspection by a human expert [18]. In particular, in the case of proprietary dependencies and IoT firmware, the code often comes in binary-only form, with no access to the source code and to the development logs, which are useful information to vet such software at a large scale. While there has been progress in automating code-level backdoor detection [3], [18]–[20], the state of the art is still limited in terms of scope and level of automation, with no significant advancements in the field for more than seven years.

B. Graybox and metamorphic fuzzing

1) Graybox fuzzing: Fuzzing [10] was originally introduced [32] as a specific form of automated software testing. It aimed at finding crashes in programs like UNIX utilities, by feeding them with randomly generated test inputs. Nowadays, fuzzing is commonly understood as a more general synonym of automated program testing. During a fuzzing campaign, the program under test (PUT) is fed with a suite of test inputs produced by an *automated input generator*. In parallel, the

runtime behavior of the PUT with these inputs is analyzed for traces of possible failures by an *automated test oracle*. The uncovered failures are symptoms of defects in the PUT, to be fixed before they cause harm or get exploited in case they pose a security threat. Fuzzing tools (or *fuzzers*) mainly differ by the way their input generators and test oracles work. In recent years, there has been a surge of interest in a specific family of fuzzers, called *graybox fuzzers*, which have been shown to be successful for security vulnerability detection [14]. **Graybox input generators** are typically based on the principles of search-based testing [11], where the PUT's input space is explored using search heuristics, designed to maximize the part of the PUT code covered by the selected inputs. Compared to pure random input generators (blackbox fuzzing) and to those based on precise code analyses (whitebox fuzzing, a.k.a. symbolic execution [33]), graybox generators aim at finding a sweet spot between (1) the ease and speed at which inputs can be generated, and (2) the ability to generate inputs achieving a high coverage of the PUT codebase. **Graybox oracles** usually rely on a lightweight mechanism, like crash detection, possibly coupled with code assertions in the PUT and sanitizers [13].

2) The AFL++ graybox fuzzer: American Fuzzy Lop (AFL) and its community-maintained successor *AFL++* [14], [34] are some of the most used and forked graybox fuzzers. They rely on a *mutation loop* that generates new inputs by randomly mutating (i.e., slightly modifying) some of the inputs generated during the previous iterations. More precisely, as the newly generated inputs are fed to the PUT, those that improve the coverage of neglected parts of the PUT code base are saved as *seed inputs* (or *seeds*). Only those seeds are then considered for mutation in the next iterations, possibly exploring even more the neglected parts of the PUT. The loop is bootstrapped with user-provided initial seeds, on which the first mutations are performed. In order to gather the code coverage data needed to guide this process, AFL++ injects additional code into the PUT. This *instrumentation code* tracks and reports which parts of the PUT's *original code* are covered during a run. Concretely, coverage is measured at the level of edges in the control-flow graph (CFG) of the PUT. Each edge is represented by a corresponding byte in a dedicated *coverage map*. The instrumentation code zeroes the map at the start of the PUT execution and, each time the execution passes through a given edge, it increments the corresponding byte in the map.

3) Binary fuzzing with AFL++: AFL++ usually performs instrumentation while compiling the source code of the PUT, via a special compilation pass plugged into a mainstream compiler. This pass injects the needed instructions in each PUT basic block, enabling the aforementioned edge coverage tracking. Yet, in this work, we aim at using AFL++ for detecting backdoors, which often requires vetting binary-only programs. For such situations, AFL++ now provides a slower binary fuzzing mode, where the PUT is run in an emulator—such as QEMU [35]—since injecting the instrumentation directly into the binary is hard to perform robustly and accurately. The emulator tracks the coverage data on the fly, by approximating edges as jumps between the memory addresses in which it has loaded the PUT code.

4) Guessing magic bytes with AFL++: A historical limitation of graybox fuzzers is their difficulty to generate inputs able to traverse specific types of branching conditions, called *magic byte comparisons*. An example of such a condition is

```
if (input[3] == 0xdeadbeef) { /* Some code here */ }
```

where a part of the input is compared to the magic byte value `0xdeadbeef`. An old enough version of AFL++ would struggle to find inputs getting past this `if` condition, as it would have to come up with the magic byte value by a series of random mutations from the initial seed values. This becomes highly unlikely to achieve in reasonable time for long enough magic byte sequences, making the fuzzer unable to test the whole part of the PUT code inside the `if` condition. This is an important problem in the context of this work, as entire classes of backdoors—such as hard-coded credentials—may rely on a magic-byte comparison— e.g., with the hard-coded username or password—as a trigger. However, recent techniques that involve splitting multi-byte comparisons into single-byte ones [16], or matching the target bytes using lightweight taint tracking [17], enable fuzzers to traverse magic byte comparisons much more efficiently. These techniques have been bundled into more recent versions of AFL++, making them credible candidates to detect backdoors.

5) Metamorphic oracles: AFL++ relies on crash detection as its main oracle mechanism. As well-coded backdoors should not cause a crash when triggered by their key input value, they cannot be detected with such an oracle. A more sensitive oracle should thus be devised for AFL++ to perceive their triggering, and developing such an oracle is a core contribution of this work. Several families of sophisticated oracles [12] have already been proposed to detect complex forms of bugs and vulnerabilities. One such family is *metamorphic oracles* [21], based on *metamorphic relations* that are expected to hold between pairs of inputs to a PUT. The principle of the oracle is then that any pair of generated inputs found violating the metamorphic relation is a trace of a PUT failure, to be further investigated. The oracle developed in this work to detect the triggering of backdoors is a form of a metamorphic oracle. Known successful uses of metamorphic oracles in fuzzing have notably enabled detecting intricate logic bugs in various complex, mature and large programs, like compilers, SQL database management systems and SMT solvers [36]–[40].

III. MOTIVATING EXAMPLE

A. A "hard-coded credentials" backdoor in sudo

The *sudo* Unix command-line tool [41] enables executing a given command as a different (usually more privileged) user. For example, `echo PASSWORD | sudo -S -u alice CMD`, when run by an entitled user `bob`, allows them to run command `CMD` as user `alice`, provided that `PASSWORD` is the correct password for user `bob`. If the password is indeed correct, sudo issues system calls to create a child process owned by `alice`, in which it executes `CMD`. Otherwise, sudo issues system calls to print an error message on the screen.

```
1   int verify_user(const struct sudoers_context* ctx
2    ,const char* password)
3   {
4     int ret = ctx->verify(password);
5     // --- Beginning of backdoor ---
6     if (strcmp(password, "let_me_in") == 0)
7       {  ret = AUTH_SUCCESS;  }
8     // --- End of backdoor ---
9     return ret;
10  }
```

Listing 1. Example of a "hard-coded credentials" backdoor in sudo.

Let us now imagine that an attacker has injected the code-level backdoor from Listing 1 in sudo. This backdoor relies on a hard-coded credentials trigger (lines 5–8) which overwrites the result of the password and entitlement check from line 4. Regardless of which user executes sudo, impersonation will always succeed if they enter the password `"let_me_in"`. This gives the attacker (and anyone informed of the key) full root access in any system containing the backdoored sudo.

B. Detecting the backdoor with ROSA

In order to understand how ROSA detects backdoors with a graybox fuzzer, we need to introduce the notion of *input families* of a PUT. Intuitively, the input values of a PUT can be classified into different families, where each family is a set of input values considered as similar in the PUT's specification, so that they result in close-by execution paths being taken in the PUT and similar effects on the PUT's environment. ROSA then introduces a *new metamorphic oracle*, relying on a new metamorphic relation, whose violations will be considered signs of possible backdoor presence: if two input values belong to the same input family, running the PUT on either of them should produce a similar effect on the PUT environment.

Let us illustrate how input families enable detecting backdoors in the sudo backdoor example. First, consider that **echo** PASSWORD | **sudo** -S -u alice CMD is run by the user bob with a fixed CMD value, so that the only actual input is the value of PASSWORD. In this restricted context, sudo has two input families: one where PASSWORD is a correct password and one where it is not, leading to two different effects on the environment (either CMD is executed as alice or an error message is printed). Second, let us assume that the effect of a program on its environment can be observed by recording the set of *system calls* that it issues. The rationale for this is that the PUT's interactions with the environment must be mediated by the operating system, which is achieved via system calls.

We then use the metamorphic oracle to detect the backdoor as follows. First, record the system calls issued by sudo when fed an incorrect password value. Then, fuzz sudo with only incorrect password values and compare the issued system calls with the recorded ones. If a significant difference is spotted, then the metamorphic relation for the family of incorrect passwords is violated and a potential backdoor is reported. This allows to detect the `"let_me_in"` password, as the execution of CMD in a child process will trigger different system calls than printing an error message. Recall that modern fuzzers have

mechanisms to quickly discover hard-coded "magic" values such as `"let_me_in"`, as detailed in Section II-B4.

In practice, all backdoors collected to build our ROSARUM benchmark also result in divergent system calls when triggered, because that is the only way for them to perform a meaningful task in the PUT's environment, no matter how small it may appear. As a consequence, all of them could possibly be detected by the metamorphic oracle described above. Yet, in order to enable such a detection, the oracle should not only be used on a single input family but on all input families deemed important enough to be searched for backdoors. However, most PUTs have numerous different input families. In the case of sudo, if we do not restrict the considered inputs to the password only, but also consider the impersonating and impersonated users, the command to be executed and the many flags that can be activated, we would end up with a combinatorial explosion of the number of families to individuate (manually) and then fuzz. To solve this issue, ROSA does not assume any prior knowledge of the PUT's input families, but instead relies on a heuristic method to automatically identify, for whatever input generated by a fuzzer, another input that should belong to the same family. The system calls issued when running the PUT with the two inputs are then compared to detect the presence of a backdoor.

IV. THE ROSA APPROACH

A. General overview

The ROSA approach to fuzzing-based backdoor detection unfolds in two successive phases, followed by a post-processing step, schematized in Figure 1. (1) During the *representative inputs collection* phase, we use a fuzzer on the PUT, in order to populate a database with a wide range of inputs that are deemed representative of its input families. (2) Then, during the *backdoor detection* phase, we fuzz the PUT for a much longer time. For every input generated by the fuzzer, we search the database of family-representative inputs, in order to heuristically identify another input that should belong to the same family as the generated input. The system calls issued when running the PUT with these two inputs are then compared. Following the principle of our metamorphic oracle, if a dissimilarity is spotted, a possible backdoor is signaled. The two inputs and the different system calls are then deduplicated and reported to the user for expert vetting.

In phase 1 (representative inputs collection), the challenge is to guide a fuzzer towards generating as many relevant family-representative inputs as possible. While inputs in the same input family produce a similar effect on the PUT environment, they can still trigger slightly different PUT behaviors. For example, wrong passwords in sudo could trigger different input sanitizing strategies, but should all eventually result in similar system calls. Consequently, we define a set of *representative inputs of a family* as any set of inputs from the family, where each input triggers a different internal behavior, among those encompassed by the family.

In practice, the internal behavior of a program can be reasonably characterized by which edges of its CFG are
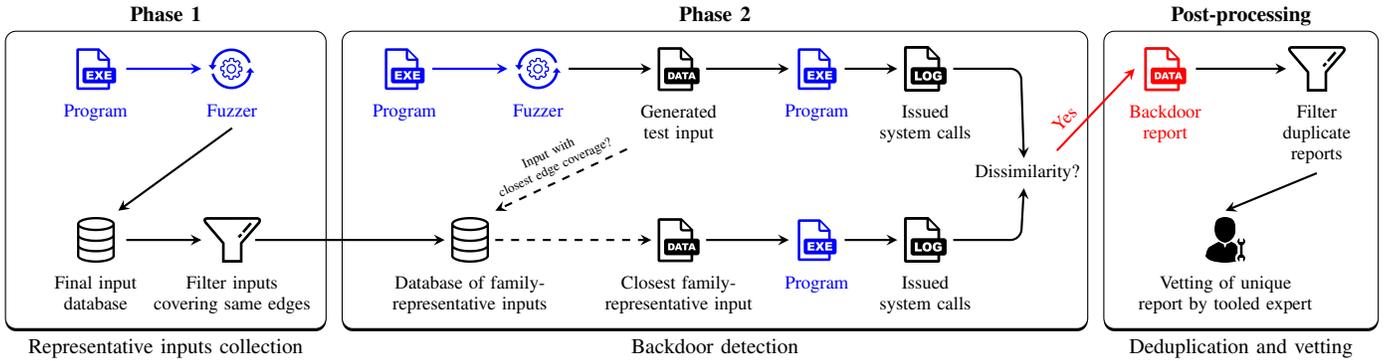
Fig. 1. General overview of the ROSA approach to fuzzing-based backdoor detection.

visited. A set of representative inputs then becomes any set of inputs from the family, where each input in the set triggers the execution of a different combination of CFG edges. In order to build our database of family-representative inputs, we thus need to identify and combine sets of inputs from each family, where all inputs in a set cover different CFG edges. This can be done by generating inputs with a fuzzer, identifying the corresponding family of each input, and adding it to the database only if the corresponding family set in the database does not already contain an input covering the same CFG edges. Yet, by the definition of input families, two inputs covering exactly the same CFG edges also belong to the same family. Hence, two distinct family sets in the database will never contain inputs covering exactly the same CFG edges and our procedure to populate the database can be performed without prior knowledge of what the input families are. One can indeed just fuzz the PUT and retain all of the generated inputs that uncover *new* CFG edge combinations in the database.

During phase 2 (backdoor detection), the main challenge is to try and identify which inputs from the representative inputs database are the most likely to belong to the same family as the inputs generated by the fuzzer. We do this by searching the database for the input that covers the closest combination of CFG edges, compared to each fuzzer-generated input. Here again, the rationale is that an input family basically defines a class of slightly different internal PUT behaviors, each best characterized by which CFG edges are visited.

B. Phase 1: representative inputs collection

During phase 1, we fuzz the PUT and populate the representative input database with all of the generated inputs that have uncovered new CFG edges. In practice, we do this by taking advantage of the existing features of common graybox fuzzers that rely on edge coverage for guidance (like the many fuzzers based on AFL). After fuzzing, we collect the seed inputs produced by the fuzzer and filter out redundant ones, as some seeds may still cover the same combination of edges.

The main stake of populating the representative inputs database in phase 1 is to properly sample the legitimate input families that will be met at phase 2. We call *family subsampling* the situation where only some of these families would be

discovered by the fuzzer in phase 1. In this case, no representative input is produced for the undiscovered families. These representative inputs would thus be missing when searching the database in phase 2. This will increase the risk of family misidentification, leading to false positives being reported, as the metamorphic oracle may mistakenly report some inputs as triggering a backdoor. We call *backdoor contamination* the situation where the fuzzer ends up triggering a backdoor in phase 1, despite the fact that backdoors are designed to be activated by a small fraction of inputs, and are thus unlikely to trigger in phase 1. In case of backdoor contamination, backdoor-triggering inputs may end up recorded incorrectly as representative inputs of legitimate families, and used for family identification in phase 2. This will increase the risk of the metamorphic oracle mistakenly classifying some inputs as legitimate. Yet, backdoors can usually be triggered by inputs belonging to many different input families (for example, the sudo backdoor from Listing 1 can be triggered with many different combinations of command-line flags) and, as backdoor contamination is unlikely to occur for *all* these families all at once, the backdoor would still be detectable with inputs from the uncontaminated families.

In practice, the levels of family subsampling and backdoor contamination can be controlled by adjusting the duration of the phase-1 fuzzing campaign. The longer the campaign, the lower the risk of family subsampling and the higher the risk of backdoor contamination. In Section VI, we perform a parameter sweep study for campaign durations between 30 seconds and 20 minutes, showing that all durations provide acceptable results on our ROSARUM backdoor benchmark.

C. Phase 2: backdoor detection

During phase 2, we fuzz the PUT again with a graybox fuzzer, but this time for as long as possible (i.e., as available resources permit, like in traditional fuzzing). For each input generated by the fuzzer, we retrieve the input that covers the closest combination of CFG edges from the representative inputs database. We run the PUT with these two inputs, and signal a backdoor if they do not trigger similar system calls.

Example. Let us imagine that the graybox fuzzer has generated an input called **Input #1**. We run the PUT on it and record which CFG edges are covered and which types

of system calls are used (among all those allowed by the operating system API, like `read`, `kill`, `open` and others in Linux). The results can be stored in vectors as follows (where **Input #1** covers the four edges of the PUT and uses all three available system call types, but `kill`):

Input #1	CFG edges	1	2	3	4
		✓	✓	✓	✓

	System calls	read		kill		open	
		✓		✗		✓	

Let us now retrieve the input that covers the closest combination of CFG edges from the representative inputs database. Let us imagine that this database contains two inputs (**Input #A** and **Input #B**), whose covered edges and used system call types are as follows:

Representative Inputs Database

Input #A	CFG edges	1	2	3	4
		✓	✗	✓	✗

	System calls	read	kill	open
		✗	✗	✓

Input #B	CFG edges	1	2	3	4
		✗	✓	✗	✗

	System calls	read	kill	open
		✗	✗	✗

To establish which of **Input #A** and **Input #B** has the closest combination of CFG edges compared to **Input #1**, we compute the Hamming distance [42] between the CFG edge vectors and consider the one for which the distance is the smallest. In this case, **Input #A** is retrieved, as its coverage w.r.t. **Input #1** differs only by two edges, against three for **Input #B**:

	1	2	3	4
Input #1	✓	✓	✓	✓
Input #A	✓	✗	✓	✗
Hamming distance for edges = 2				

	1	2	3	4
Input #1	✓	✓	✓	✓
Input #B	✗	✓	✗	✗
Hamming distance for edges = 3				

Finally, system call comparison is done by computing the Hamming distance between the system call type vectors of **Input #1** and the retrieved input **Input #A**:

	read	kill	open
Input #1	✓	✗	✓
Input #A	✗	✗	✓
Hamming distance for system calls = 1			

We report a backdoor as soon as this distance is non-zero. In this case, a backdoor is reported because **Input #1** uses a `read` system call and **Input #A** does not. Note that if, during database retrieval, the Hamming distance between CFG edge vectors had been the same for **Input #A** and **Input #B**, we would have reported a backdoor only if the Hamming distance between the system call type vectors had been non-zero for *both* **Input #A** and **Input #B**.

D. Post-processing: deduplication and vetting

Fuzzers are stochastic and thus subject to repeatedly triggering the same issues in the PUT. As a consequence, they often use deduplication techniques, to try and avoid polluting their output with duplicated reports (see Donaldson et al. [43] for an example). ROSA follows a similar approach and reports a backdoor only if it had not previously reported another backdoor involving the same system call difference with the same family-representative input, as these two reports are likely to describe the same backdoor. Our experiments show that this heuristic enables reducing by one order of magnitude on average the number of reports produced by ROSA, without impeding its ability to find backdoors.

As family identification is made heuristically, ROSA is capable of returning false positives. Each backdoor report (of the form "**Input #1** is suspicious because it uses a `read` system call and **Input #A** does not") must thus be manually vetted by an expert (an understanding of the expected and actual program behaviors is indeed needed here, for which there is currently no fully-automatic solution). Yet, vetting can still occur in a pretty systematized and semi-automated way:

1) Run the PUT under a process tracing tool (like `strace` [44]) with both reported inputs, filtering out all system calls except the ones listed in the report;
2) Compare the filtered traces to determine whether one represents a privilege escalation within the program or an undue access to the underlying system.

Example. Let us illustrate how vetting works on a real (positive) report returned by ROSA on the authentic ProFTPD backdoor [3]. Running the suspicious input under `strace` and keeping only the divergent system calls produces the following pattern: `clone3(...)`, `setuid(0)`, `setgid(0)`, `execve("/bin/sh", ...)`, which is a transparent attempt at spawning a root shell (not an expected behavior for ProFTPD). The suspicious input itself, 14-lines long, contains the surprising `HELP ACIDBITCHEZ` command, which can be identified as part of the key input value of a backdoor.

V. IMPLEMENTATION

We have implemented the ROSA tool by relying on AFL++ [14] (version `++4.20c`) for the fuzzing campaigns of both phases 1 and 2. The ROSA tool needs to be provided with the PUT and a corpus of initial seed inputs to fuzz it. Its main parameters are the respective lengths of the two fuzzing campaigns. At the end, the tool returns a list of PUT inputs that trigger potential backdoors, similar to how vanilla AFL++ returns a list of crash-triggering inputs. These inputs can then be vetted as discussed above.

As backdoor detection must often be performed on PUTs that come in binary-only form, we use AFL++ in binary-only mode [15], with QEMU [35] as backend emulator. Otherwise, we configure AFL++ by following the best practices described in its documentation. This notably means using six synchronized instances of the fuzzer running in parallel, with different seed prioritization and mutation strategies. Half of the instances only instrument the PUT itself, while the other half also instrument the called external library functions, enabling backdoor detection in dynamically loaded libraries. All instances leverage the built-in AFL++ mechanisms to deal efficiently with magic byte comparisons [16], [17], as these are often used as backdoor triggers. It should be noted that support for the configuration of AFL++ that we have just described is

currently limited to fuzzing x86/x64 binaries on Linux. Our tool inherits thus this limitation.

Finally, during the fuzzing campaigns of phases 1 and 2, our tool records on the fly which CFG edges and system calls are triggered by the generated inputs, for later use according to the ROSA approach. This is implemented through light modifications to the AFL++ and QEMU code. For recording edge coverage data, we rely on the existing edge coverage measurement mechanisms of the fuzzer. For recording system call coverage data, we implement similar mechanisms to those used for edge coverage measurement, but we track system call instructions instead of jump instructions between basic blocks.

VI. EXPERIMENTAL EVALUATION

A. General overview

We aim at answering the following research questions:

RQ1 Can ROSA detect backdoors in enough diverse contexts, with enough robustness, speed and automation, to make it usable and useful in the wild?

RQ2 How does ROSA compare to state-of-the-art backdoor detection tools, in terms of robustness, speed and automation?

To answer these, we need to run ROSA and competing tools over samples of backdoors in real programs. Yet, there is no existing off-the-shelf backdoor dataset that could be leveraged to do so. Indeed, previous papers introducing program analysis tools for backdoor detection [3], [18]–[20] use different small backdoor datasets for evaluation purposes. In addition, these papers date back 7–11 years, so that some samples have been lost, while those that are still available can be undocumented binary firmware, running only on old IoT devices. As a preliminary step to our evaluation, we have thus *assembled a novel backdoor benchmark dataset, called* ROSARUM.

B. Constructing the ROSARUM benchmark

1) Collecting and porting authentic backdoors: As a first step to populate ROSARUM, we have collected authentic backdoors from three types of sources. First, we have looked into all state-of-the-art papers [1]–[3], [18]–[20] for alive references to backdoor samples. When necessary, we have also contacted their authors to verify the availability of the backdoors mentioned in the papers. Second, we have searched public vulnerability and exploit databases for sufficiently documented reports that describe genuine code-level backdoors. Finally, we have searched general and IT news reports (gray literature) for references to code-level backdoors, and looked for the corresponding backdoor samples. In total, we have performed an in-depth sample search and analysis for 15 backdoor reports, including 11 from the state of the art, 1 from public databases and 3 from gray literature. Samples could not be obtained for 4 reports from the state of the art. As the ROSA tool only supports x86/x64 Linux binaries, we had to make sure that collected backdoors work on such a platform. While 5 samples already supported it natively, the 6 remaining ones did not. We invested significant effort in porting them. The RaySharp and QSee backdoors from the state of the art [18]

are the only ones that could not be ported, due to unresolved dependencies on libraries and IoT peripherals.

2) Seeding synthetic backdoors in a fuzzing benchmark: To further enrich ROSARUM with more diverse programs and backdoors, we have followed an approach used in the state of the art [3], where students were asked[1] to inject synthetic backdoors into the ProFTPD program. We have hence asked a researcher *not involved in the development of* ROSA (in the style of clean-room design) to seed synthetic backdoors into the programs of MAGMA [22], a recent fuzzing benchmark. These programs are "open-source libraries with widespread usage and a long history of security-critical bugs" [22]. They are deemed challenging for modern fuzzers, so they are used to evaluate their vulnerability detection capabilities. The ROSA developers performed blind detection campaigns over these programs, with no knowledge of the backdoors' inner workings.

3) Preparing the backdoors for benchmarking: For every authentic or synthetic backdoor in ROSARUM, we include the safe source of the program and two patches, a *backdoor patch* and a *ground-truth patch*. The backdoor patch simply injects the backdoor into the program source, while the ground-truth one inserts backdoor detection markers instead. These markers print a predefined string to denote the successful triggering of the backdoor, enabling one to verify if suspicious inputs reported by detection tools are true/false positives/negatives. Once a patch possibly applied, the source can be compiled into a x86/x64 Linux binary using the provided Makefile.

4) Description of the final benchmark: In total, ROSARUM contains 17 backdoors (7 authentic + 10 synthetic, i.e. the largest benchmark ever used to evaluate backdoor detection techniques, as per the state of the art), detailed in Table I. The sudo backdoor of Listing 1, used earlier to illustrate the ROSA approach, is also included. On the other hand, the authentic Trendnet backdoor described in the state of the art [18] is not included, as our analysis revealed that it was in fact a false positive, due to inoffensive hard-coded credentials. We have also excluded the recent authentic xz [5] backdoor, as our analysis revealed that triggering the backdoor requires passing a signature check with a hard-coded public key, so that only an attacker knowing the private key can do it. This mechanism prevents triggering the backdoor with a fuzzer or other dynamic analyses in reasonable time, as they need to brute-force the cryptography[2]. We elaborate further on this in Section VI-C, together with other limitations of ROSA.

C. RQ1: usability and usefulness of ROSA

1) Experimental protocol: We design our experimental protocol by carefully adapting the best practices for fuzzing evaluation [45], [46] to the backdoor case, for which no baseline exists. Specifically, we perform 10 independent 8-hour runs of ROSA for each backdoor in ROSARUM. We

---

[1]Automatic injection of non-trivial backdoors in random code is a hard problem, and there exists no automatic backdoor injection tool to date.

[2]The xz backdoor is particularly intricate and adversarial. It also includes dynamic code modifications that are hard to handle for static analyses.

TABLE I

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

allocate 8 CPU cores and 16 GiB of RAM to each run, on a dedicated Intel® Xeon® Silver 4241 2.20 GHz server. For the MAGMA [22] programs, we use the initial seeds provided by the benchmark; we use the standard seeds from the AFL++ documentation for HTTP[3] and FTP[4] servers; we use a single seed containing the string `"test"` for the remaining programs.

We evaluate two different aspects of ROSA. First, we **measure the robustness and speed of ROSA**, in diverse contexts and at scale. To do so, we measure, for each ROSARUM backdoor, (1) the proportion of ROSA runs that fail to find the backdoor before timing out, and (2) the minimum, average, and maximum time (over the 10 runs) needed by ROSA to find a first input that triggers the backdoor. Second, we **measure the level of automation of ROSA**, i.e. we evaluate how much additional manual work is required by ROSA, compared to using a classical fuzzer like AFL++ for finding crashes. The main overhead of ROSA comes from its ability to produce false positives, while crash reports are usually legitimate. An expert must thus inspect all of the reported inputs one by one, until they discover the backdoor or discard all of the inputs as false positives, following the semi-automated procedure described at Section IV-D. We estimate this manual expert effort by reporting, for each ROSARUM backdoor, the minimum, average and maximum number of inputs that an expert should draw at random, either to have a 95% probability of coming across an input that triggers the backdoor (for the runs that succeed in finding it) or to establish that no input triggers it (for the runs that fail). We also report on the time needed to vet an input with our semi-automated procedure.

2) Duration of phase 1: As discussed in Section IV-B, ROSA is parameterized by the duration of the fuzzing campaign at phase 1 and this parameter can impact both the robustness and manual effort of the approach. To evaluate
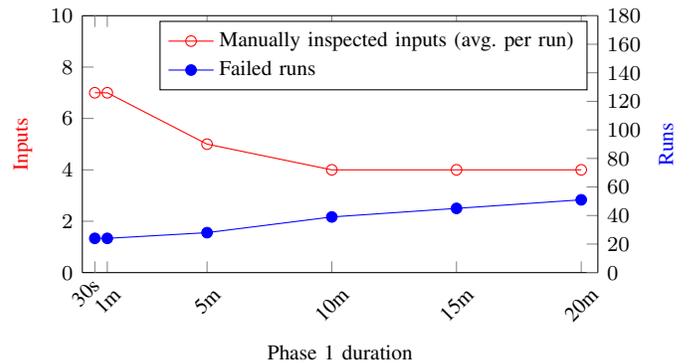
Fig. 2. ROSA parameter sweep study for the duration of phase 1. For each duration, we performed a total of 180 runs of 8 hours (10 runs per backdoor in ROSARUM, including with specialized seeds for the Belkin backdoor).

the sensitivity of ROSA to this parameter and pick an optimal value, we have performed a parameter sweep study for durations between 30 seconds and 20 minutes. Within this range, we have measured the average number of manually inspected inputs and the total number of failed runs, for all backdoors in ROSARUM. The results are detailed in Figure 2. As expected, the number of inspected inputs decreases and the number of failed runs rises when longer durations are used, as this reduces family subsampling but also increases backdoor contamination. In the worst cases, ROSA still remains useful, as only 7 inputs must be analyzed manually, while 72% of the runs still succeed. As we value better detection capabilities over lower manual effort, we choose 1 minute as the optimal value and it is the one used for collecting the detailed ROSA evaluation results discussed in the next paragraphs.

3) Results discussion: The detailed ROSA evaluation results are presented in Table II.

In terms of **robustness**, ROSA was able to discover the backdoor during 156 (87%) of all 180 performed runs. For 11 (65%) of the 17 backdoors, all runs were successful. For

Backdoor	ROSA — (10 runs × 8 hours) / backdoor — 1 minute of fuzzing for phase 1								STRINGER	
	Robustness + speed				Automation level				*Backdoor detection time*	*Manually inspected strings*
	Failed runs	*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	Not found	**0**
+ with specialized seeds*	**3 / 10**	**17m40s**	**3h49m29s**	Timeout	2781	4	5	7	Not found	**0**
D-Link / thttpd	**0 / 10**	**2m07s**	**15m00s**	**43m42s**	3648	**7**	**9**	**12**	Not found	113
Linksys / scfgmgr	**0 / 10**	**1m05s**	**1m29s**	**1m55s**	251	1	1	1	Not found	**0**
Tenda / goahead	**0 / 10**	**1m28s**	**3m34s**	**8m10s**	535	**1**	**2**	**2**	Not found	290
PHP	1 / 10	24m30s	2h03m44s	Timeout	11631	**4**	**8**	**16**	**6m**	573
ProFTPD	4 / 10	4m03s	3h37m32s	Timeout	2995	**5**	**8**	**11**	**7s**	314
vsFTPd	**0 / 10**	**3m04s**	**5m41s**	**11m03s**	1890	**3**	**4**	**4**	Not found	117
Synthetic backdoors										
sudo	**0 / 10**	**5m47s**	**8m05s**	**11m46s**	167	**1**	**1**	**1**	Not found	137
libpng	2 / 10	13m47s	2h24m46s	Timeout	4202	**1**	**2**	**2**	**4s**	9
libsndfile	3 / 10	2h21m08s	5h04m46s	Timeout	10376	**9**	**12**	**13**	**5s**	8
libtiff	**0 / 10**	**5m08s**	**12m15s**	**25m10s**	9566	**1**	**3**	**5**	Not found	31
libxml2	**0 / 10**	**8m17s**	**27m14s**	**1h09m06s**	12104	**9**	**14**	**20**	Not found	1208
Lua	1 / 10	**50m34s**	**4h07m41s**	Timeout	6653	**6**	**12**	**17**	Not found	36
OpenSSL / bignum	**0 / 10**	**9m53s**	**22m00s**	**39m52s**	1441	**1**	**1**	**2**	Not found	657
PHP / unserialize	**0 / 10**	**23m05s**	**1h04m39s**	**1h35m08s**	6285	**1**	**1**	**1**	Not found	974
Poppler	**0 / 10**	**11m28s**	**49m09s**	**1h33m02s**	9544	**5**	**6**	**8**	Not found	543
SQLite3	**0 / 10**	**33m17s**	**1h02m52s**	**2h42m42s**	4705	**20**	**26**	**31**	Not found	226

* Two variants of initial fuzzing seeds were used for Belkin: unspecialized (*U*) and specialized (*S*) ones. Variant *U* are the default AFL++ seeds for HTTP servers, with which the backdoor could never be triggered by AFL++ in 10 runs of 8 hours. Variant *S* are specialized seeds, targeting the URL parser of the server, with which the backdoor was triggered in 7 of the 10 AFL++ runs. The oracle could always recognize the backdoor, once AFL++ had triggered it.

17 (100%) of them, at least half of the runs were successful. A single backdoor was never detected by ROSA using the default seed corpus: the Belkin backdoor. Yet, with specialized seeds (bare GET requests, specifying only a target URL), aiming at the URL parser of the analyzed HTTP server (as URLs are a convenient place to hide backdoor keys), ROSA detected the backdoor in 7 out of 10 runs, showing that targeting exposed parts of the input space can help accelerate backdoor detection.

Experimental results show that the ROSA oracle is robust, with a very low rate of false negatives. A backdoor miss can be caused either by a fuzzer miss (AFL++ does not generate any input triggering the backdoor during the run) or a ROSA oracle miss (AFL++ does generate such inputs, but they are not recognized by the ROSA oracle, i.e., false negatives). By relying on the ROSARUM ground truth, we were able to establish that all of the 24 failed runs (100%) were caused by a fuzzer miss and none by a ROSA oracle miss. In particular, the backdoor contamination effect discussed in Section IV-B had no influence on detection capabilities in practice. Backdoor contamination at phase 1 occurred only during 5.56% of all 180 runs. Moreover, in none of these runs were the backdoor-triggering inputs from phase 2 always matched up with contaminated inputs triggering identical system calls, meaning that backdoor detection was never prevented by contamination.

The main limiting factor for ROSA robustness is the best-effort nature of fuzzing. Like with all dynamic analyses, detection is limited to the inputs that the fuzzer can test with available resources. All of the well-known obstacles to fuzzing will increase the probability of a backdoor miss. These include large input spaces, complex input formats, slow execution times, and hard-to-inverse operations, like hashing or cryptography (as in the xz [5] backdoor). How combining ROSA with static analyses could help alleviate the impact of these obstacles (e.g., through slicing [47]), or identify suspicious obstacles, are interesting directions for future work.

In terms of **speed**, the average time to the first backdoor detection across all 156 successful runs is 1h30m35s. 9 backdoors (53%) can be detected by ROSA in less than one hour on average; 16 backdoors (94%) in less than five. Here again, the detection speed of ROSA is principally determined by the velocity at which the fuzzer can trigger the backdoor. The observed detection times are in line with those of classical fuzzing, where AFL++ aims for crash-triggering bugs.

In terms of **automation level**, for 4 (24%) of the 17 backdoors, at most one input must be manually inspected on average; for 13 (76%) of them, at most 10 inputs; for the 4 (24%) remaining backdoors, a maximum of 31 inputs had to be manually inspected. These results demonstrate the selectivity of our metamorphic oracle, which cuts by two to four orders of magnitude the number of vetted inputs, compared to the baseline detailed in Table II, where all AFL++ seeds would have to be vetted. Our experiments also reveal that the semi-automated backdoor vetting procedure from Section IV-D is effective and requires about 2 minutes of manual effort per input on average. We conclude that, for most backdoors, ROSA has an either negligible or moderate manual overhead (from 2 minutes to 15 minutes) compared to classical fuzzing.

Answer to RQ1 (usability and usefulness)

ROSA **detects all backdoors** from our benchmark with a **level of robustness and speed similar to traditional fuzzing**. ROSA also has a **level of automation similar to traditional fuzzers**, but produces false positives that have to be manually discarded. Yet, the required manual effort is low, and limited to vetting an average of 7 suspicious runtime behaviors on our benchmark.

D. RQ2: comparison with the state of the art

1) Availability of competing tools: To the best of our knowledge, the state of the art in program analysis for backdoor detection consists of four tools (2013–2017): WEASEL [3], FIRMALICE [19], STRINGER [18], HUMID-IFY [20]. After verification and communication with authors, it appears that only STRINGER is still both available and working on modern systems. We compare with STRINGER below and discuss the other three as a part of related work (Section VII).

2) Comparison with STRINGER: STRINGER statically analyzes a binary program and extracts a list of statically-coded strings, likely to be part of a backdoor trigger. We compare STRINGER with ROSA on the ROSARUM benchmark. Among the three case studies from the STRINGER paper [18] that lead to the detection of hard-coded credentials, we could not include in ROSARUM the RaySharp and QSee backdoors, as they affect firmware from very old CCTV and DVR devices, which we could not make run on modern systems. While we were able to run the code involved in the third case study (TRENDnet firmware), we did not include it in ROSARUM either, as manual analysis revealed that it was actually not a backdoor, but just a hack to deal with user-defined credentials configured to be empty (the STRINGER paper refers to it not as a backdoor but as "additional functionality"). As STRINGER was also used to recover the FTP command set from a safe version of vsFTPd, we included a different version of this program, infected by an authentic backdoor from another source [4], into ROSARUM.

For each backdoor, we report in Table II (1) whether a part of the backdoor trigger was detected, (2) how long the detection took and (3) how many strings had to be manually inspected before making a decision about backdoor presence.

In terms of **robustness**, only 4 backdoors out of 17 (24%) can be detected by STRINGER, compared to 17 (100%) with ROSA. In particular, only 2 out of the 7 authentic backdoors from ROSARUM could be detected by STRINGER (vs 7 for ROSA) and the vsFTPd backdoor was detected by ROSA but not by STRINGER. These shortcomings are due to STRINGER relying on imprecise heuristics. These hypothesize that backdoor triggers should involve static strings with specific properties, which is not true for most non-trivial backdoors. In terms of **speed**, STRINGER is 1,928 times faster than ROSA on average. This is due to STRINGER relying on a simple static analysis, where ROSA relies on brute-force dynamic analysis. In terms of **automation**, STRINGER requires the manual inspection of 308 strings on average, compared to 7 inputs for ROSA. The manual inspection time per unit appears more important with STRINGER, as the expert must reverse-engineer the (binary) program to locate each string and evaluate its dangerousness.

Answer to RQ2 (comparison with competing tools)

Out of the four existing program analyzers for backdoor detection, **only STRINGER is available and working**. It relies on a simple static analysis that **cannot detect most backdoors from our benchmark**. STRINGER identifies the few detected backdoors way **faster than ROSA** but returning **44 times more, harder-to-vet false positives**.

E. Threats to validity

A first class of threats to the internal validity of our answers to the research questions arise because of *possible defects in the software artifacts and manual operations* that we have relied on. However, AFL++ is a popular, community-maintained and open-source fuzzer, which is employed as a baseline in many fuzzing papers. Our ROSA layer and our experimental infrastructure have been tested at unit level and on small-scale fuzzing campaigns. Our ROSARUM benchmark has been tested by one developer and fuzzed by another one. The results of our manual vetting campaigns have been cross-checked with the ground-truth provided by ROSARUM. A second class of threats to internal validity is that our experimental results might not be significant, due to the *highly random nature of the fuzzing process*. However, we have mitigated this threat by averaging the results over 10 independent 8-hour runs. In addition, we have systematically discussed the robustness of the ROSA oracle and underlying fuzzer.

Common to all empirical studies, this one may be of *limited generalizability*. To reduce this threat, we have performed our experiments over the 17 backdoors from the ROSARUM benchmark. This includes 7 authentic backdoors, obtained after painstaking collection and porting efforts, and 10 synthetic backdoors, injected into a standard fuzzing benchmark, in the style of clean-room design. In particular, several of these backdoors trigger only system calls as common as benign inputs do, but are still detected by ROSA, showing that the approach is not limited to detecting exotic behaviors. While one cannot rule out the possibility of an advanced backdoor trigger escaping detection by the ROSA oracle, ROSA would still significantly raise the bar for attackers, likely forcing them to spend extra efforts building more intricate (and thus more noticeable) backdoors.

VII. RELATED WORK

Backdoors. Code-level backdoors induce deviant behaviors that introduce vulnerabilities in software. Some previous works considered backdoors in other (non-software) parts of computer systems, while others investigated different kinds of malicious or exploitable deviant behaviors in classical software. The first class of works covers taming backdoors in hardware [48], cryptography [49], and machine learning [50]. The second includes detecting malware [51] (malicious active

programs), as well as exploitable bugs, with a significant part of fuzzing research focusing on detecting memory bugs in memory-unsafe languages [10], which pose security threats.

Exploitable bugs can serve a similar purpose as backdoors. So-called "bugdoors" [1] are bugs injected *on purpose* in programs, for later exploitation by informed attackers. Bugdoors can be more plausibly denied by their authors than backdoors, but they can be detected using well-established tools and practices for software hardening. Bug detection with fuzzing usually relies on simple oracles, like crash detection [14], [34] and sanitizers [13], but metamorphic oracles have also been coupled with fuzzers to find logical bugs in code processors or constraint solvers [36]–[40].

Detection of code-level backdoors. Research on code-level backdoors has been rather scarce [1]. Four approaches and corresponding tools have been proposed to analyze (binary) program code and detect backdoors. We have discussed STRINGER and compared it experimentally to ROSA in Section VI-D. WEASEL [3] aims at detecting authentication bypass (e.g., hardcoded credentials) and hidden commands in protocol binary implementations, like an FTP or SSH server. WEASEL tests the binary with inputs generated from the protocol specification and analyzes the resulting execution traces, to locate the functions or code blocks in the code that process commands or grant authentication. The system and external library calls performed in located code blocks must then be extracted with a disassembler and manually inspected for suspicious patterns. In comparison, ROSA is not restricted to authentication bypass and hidden commands and can detect backdoors in diverse kinds of programs. In addition, ROSA returns dubious inputs and the suspicious system calls that they issue, where WEASEL only returns sensitive basic blocks or functions in a binary, to be manually reverse-engineered. FIRMALICE [19] requires to be supplied with a target point in a binary program, corresponding to the execution of operations restricted to authenticated users. Symbolic execution is then applied on a backwards slice of the program to automatically discover inputs reaching the target, hinting at the possible presence of an authentication bypass in the code. Contrary to ROSA, FIRMALICE is thus limited to authentication bypass detection and requires manually reverse-engineering a binary to identify the targets. HUMIDIFY [20] provides a machine learning model, trained to infer which common protocol (like HTTP or SSH) is implemented by a binary program. The tool comes with a platform, enabling human experts to specify and verify the feature profile of such protocols (e.g., "an HTTP server uses TCP, but not UDP and may read/write files"). Given a binary program to vet, the machine learning model detects the implemented protocol and the platform verifies the binary for a divergence with its expected profile. Contrary to ROSA, HUMIDIFY is thus limited to detecting simple hidden features in binaries implementing common protocols. In particular, malicious but profile-compatible behaviors, like hard-coded credentials, cannot be detected by HUMIDIFY. Moreover, vetting if the reported profile violations are actual backdoors is likely to require reverse-engineering the analyzed

binaries. Overall, **ROSA appears to be the first program analyzer for backdoor detection that:** (1) relies on graybox fuzzing, (2) does not restrict by nature the type of backdoors that can be identified or the category of programs that can be analyzed, and (3) does not require systematic manual reverse-engineering of the analyzed binary.

Other works. The literature on code-level backdoors also includes two other works that are less relevant, but still quite complementary to ROSA. Ganz et al. [2] detect backdoor injections in collaborative development, with a machine learning model that recognizes suspicious contributions in version control systems repositories. Thomas et al. [1] present a theoretical framework to define backdoors and reason about detection. The wider literature includes works that share conceptual elements with ROSA. The two-phase approach of ROSA, where one first elucidates standard behaviors and then searches for behaviors violating these standards, is somewhat similar to the method of DIDUCE [52]. Yet, DIDUCE elucidates invariants about the values taken by the expressions of the program, to search for classical bugs. In contrast, ROSA focuses on input family elucidation to search for backdoors. Like backdoors, side-channel vulnerabilities provide a subtle way to sneak into programs, by guessing secret runtime values through observation of non-functional program behaviors. Complex test oracles were also considered to find side-channel vulnerabilities with fuzzing, like for example in DIFFUZZ [53]. Yet, DIFFUZZ relies on a differential oracle, where ROSA is based on a metamorphic oracle. In network security, backdoors often refer to maliciously modified server configurations, opening an undocumented remote access to the server. Several works have tried detecting such *network-level backdoors*, like PBDT [54], which uses machine learning to scan Python-based servers.

VIII. CONCLUSION

In this work, we have introduced ROSA, the first approach and tool that uses graybox fuzzing to detect code-level backdoors. ROSA complements a state-of-the-art fuzzer (AFL++) with a metamorphic oracle that can identify backdoor triggers at runtime. To enable the experimental evaluation of ROSA and its comparison with existing tools, we have built ROSARUM, the first publicly-available benchmark for evaluating backdoor detection tools in diverse programs. ROSARUM contains 7 authentic and 10 synthetic backdoors inserted into a standard fuzzing benchmark. The experimental evaluation of ROSA over the 17 backdoors of ROSARUM shows that ROSA has a level of robustness, speed, and automation similar to classical graybox fuzzing (for non-backdoor bugs such as crashes). Compared to the state of the art, ROSA is capable of handling a wide scope of backdoors and programs and it does not require manually reverse-engineering the investigated (binary) code.

REFERENCES

[1] S. L. Thomas and A. Francillon, "Backdoors: Definition, Deniability and Detection," in *Research in Attacks, Intrusions, and Defenses*, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds. Cham: Springer International Publishing, 2018, vol. 11050, pp. 92–113.

[2] T. Ganz, I. Ashraf, M. Härterich, and K. Rieck, "Detecting Backdoors in Collaboration Graphs of Software Repositories," in *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*. Charlotte NC USA: ACM, Apr. 2023, pp. 189–200.

[3] F. Schuster and T. Holz, "Towards reducing the attack surface of software backdoors," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*. Berlin, Germany: ACM Press, 2013, pp. 851–862.

[4] C. Evans, "Alert: vsftpd download backdoored," 2011, https://scarybeastsecurity.blogspot.com/2011/07/alert-vsftpd-download-backdoored.html [Accessed: July, 19, 2024].

[5] I. Red Hat, "Malicious code was discovered in the upstream tarballs of xz," 2024, https://nvd.nist.gov/vuln/detail/CVE-2024-3094 [Accessed: May, 22, 2024].

[6] J. Toterhi, "Hunting for backdoors in iot firmware at unprecedented scale," in *Proceedings of the 2018 Hack in the Box Dubai Hacking conference Security - HITBSecConf Dubai '18*, 2018.

[7] Z. Michael Lee, "D-link routers found to contain backdoor," 2013, https://www.zdnet.com/article/d-link-routers-found-to-contain-backdoor [Accessed: May, 22, 2024].

[8] d. Craig, "From china, with love," 2013, https://web.archive.org/web/20131020145741/http://www.devttys0.com/2013/10/from-china-with-love [Accessed: May, 22, 2024].

[9] E. Benoist-Vanderbeken, "Some codes and notes about the backdoor listening on tcp-32764 in linksys wag200g," 2015, https://github.com/elvanderb/TCP-32764/tree/master [Accessed: May, 22, 2024].

[10] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, Jan. 2020.

[11] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2009.

[12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[13] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1275–1295.

[14] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *WOOT'20: Proceedings of the 14th USENIX Conference on Offensive Technologies*, Aug. 2020, p. 10.

[15] AFL++, "Qemu-afl," 2024. [Online]. Available: https://github.com/AFLplusplus/qemuafl

[16] "Circumventing Fuzzing Roadblocks with Compiler Transformations," Aug. 2016. [Online]. Available: https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/

[17] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019.

[18] S. L. Thomas, T. Chothia, and F. D. Garcia, "Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality," in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, vol. 10493, pp. 513–531.

[19] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," in *Proceedings 2015 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2015.

[20] S. L. Thomas, F. D. Garcia, and T. Chothia, "HumIDIFy: A Tool for Hidden Functionality Detection in Firmware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, vol. 10327, pp. 279–300.

[21] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[22] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A Ground-Truth Fuzzing Benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, Nov. 2020.

[23] D. Kokkonis, M. Marcozzi, E. Decoux, and S. Zacchiroli, "The ROSA toolchain," 2025, archived on Software Heritage with SWHID swh:1:rev:d30f7f1800a5dde3b9991125f9b911f8396c6346. [Online]. Available: https://github.com/binsec/rosa

[24] ——, "The ROSARUM backdoor detection benchmark," 2025, archived on Software Heritage with SWHID swh:1:rev:21d986293f083a09c0692c504305ac6e4fb9bf38. [Online]. Available: https://github.com/binsec/rosarum

[25] V. G. Lokhande and D. Vidyarthi, "A study of hardware architecture based attacks to bypass operating system security," *Security and Privacy*, vol. 2, no. 4, p. e81, 2019.

[26] N. Kostyuk and S. Landau, "Dueling over dual_ec_drbg: The consequences of corrupting a cryptographic standardization process," *Harv. Nat'l Sec. J.*, vol. 13, p. 224, 2022.

[27] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[28] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 23–43.

[29] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1509–1526.

[30] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and Evolution of Package Dependency Networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. Buenos Aires, Argentina: IEEE, May 2017, pp. 102–112.

[31] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empir. Softw. Eng.*, vol. 24, no. 1, pp. 381–416, 2019.

[32] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[33] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USENIX Association, 2008, pp. 209–224.

[34] M. Zalewski, "American Fuzzy Lop - Whitepaper," Tech. Rep., 2016. [Online]. Available: https://lcamtuf.coredump.cx/afl/technical_details.txt

[35] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[36] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, "Skeletal approximation enumeration for smt solver testing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1141–1153.

[37] Z. Su and C. Sun, "Emi-based compiler testing," 2024. [Online]. Available: https://web.cs.ucdavis.edu/~su/emi-project/

[38] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020.

[39] A. Lascu, M. Windsor, A. F. Donaldson, T. Grosser, and J. Wickerson, "Dreaming up metamorphic relations: Experiences from three fuzzer tools," in *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*. IEEE, 2021, pp. 61–68.

[40] S. Project, "Sqlancer," 2024. [Online]. Available: https://github.com/sqlancer/sqlancer

[41] Sudo Project, "Sudo," 2024. [Online]. Available: https://www.sudo.ws/

[42] R. Hamming, *Coding and Information Theory*. Prentice-Hall, 1980.

[43] A. F. Donaldson, P. Thomson, V. Teliman, S. Milizia, A. P. Maselco, and A. Karpiński, "Test-case reduction and deduplication almost for free with transformation-based compiler testing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1017–1032.

[44] Strace, "Strace Linux utility," 2024. [Online]. Available: https://github.com/strace/strace

[45] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138.

[46] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 140–140.

[47] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[48] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Des. Test*, vol. 27, no. 1, p. 10–25, jan 2010.

[49] C. Easttom, "A study of cryptographic backdoors in cryptographic primitives," in *Electrical Engineering (ICEE), Iranian Conference on*, 2018, pp. 1664–1669.

[50] Y. Li, Y. Jiang, Z. Li, and S.-T. Xia, "Backdoor learning: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 1, pp. 5–22, 2024.

[51] O. A. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, pp. 6249–6271, 2020.

[52] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering - ICSE '02*. Orlando, Florida: ACM Press, 2002, p. 291.

[53] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "DifFuzz: Differential Fuzzing for Side-Channel Analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 176–187.

[54] Y. Fang, M. Xie, and C. Huang, "PBDT: Python Backdoor Detection Model Based on Combined Features," *Security and Communication Networks*, vol. 2021, pp. 1–13, Sep. 2021.