Weird Machines in Package Managers: A Case Study of Input Language Complexity and Emergent Execution in Software Systems

Sameed Ali
Dartmouth College
Hanover NH, USA
sameed.ali.gr@dartmouth.edu

Michael E. Locasto
Narf Industries
New Jersey, USA
michael.locasto@narfindustries.com

Sean Smith
Dartmouth College
Hanover NH, USA
sws@cs.dartmouth.edu

***Abstract*—Unexpected interactions of linguistic elements of software often produce unexpected composable computational artifacts called weird machines. Using the RPM package manager as a case study, we provide a systematic approach to discover and classify the semantics of latent functionality existing within the input space of complex systems. We demonstrate latent functionality present within the RPM package management infrastructure via the construction of a Turing-complete automaton residing within the RPM package management infrastructure.**

***Index Terms*—weird machines, exploit, software security, emergent computation**

I. Introduction

The increasing complexity of software systems often leads to unexpected computational abilities within them. These computational abilities manifest themselves as unintended composable computing primitives that remain undiscovered within software systems. Also referred to as weird machines (e.g., [1]), these primitives pose a serious risk for building secure systems because they can be utilized by an adversary to carry out unauthorized computation within a target system. In addition to subverting a user's judgment about the safety of input, the ability to carry out arbitrary computation via what appears to be benign metadata (e.g., [2]) not only greatly increases the impact of otherwise minor security vulnerabilities, but also makes it possible to realize exploits that do not leverage well-known exploitation vectors like memory corruption.

However, once a vulnerability is found that can program a weird machine, researchers then build tools that retroactively test the existence of those specific types of weird machines. Such tools, for example, might prevent ROP gadgets, return-to-libc attacks, memory corruptions (buffer overflows and the like) etc. This raises an interesting question, how can we increase assurance that software design is secure from weird machines from the start? Further, what is the relationship between input language complexity and the existence of weird machines? And how can the interaction of various features of a complex system give rise to them?

To answer these questions, we investigate the RPM package management infrastructure as a complex system. We investigate how the various components of the system interact with each other and analyze its metadata to see how it can be used to program a weird machine. By considering the RPM package management infrastructure, we investigate the possibility of taking a design or an early-state prototype and forecasting where and how computation may arise and be programmed within it. We argue that designs of sufficient complexity necessarily generate unintended programming primitives and seek to shed light on that complexity boundary.

To that end, this paper makes the following **contributions**:

- It presents a well-documented case-study of the discovery of latent functionality within the RPM package management infrastructure.
- It provides an analytical method for assessing software systems at design time to discover the presence of weird machines within them.
- It sheds light on the computational power of metadata in package managers and demonstrates how it is powerful enough to embed computation within itself.

A. Motivation

Malicious users are known to utilize unintended computational abilities in software systems for adversarial ends; Figure 1 illustrates the general model. For instance, scriptless attacks [3] have been demonstrated on web browsers, allowing an attacker to perform computation on a victim's browser without relying on memory corruption or other such vulnerabilities. Attackers in the wild have also utilized such computing artifacts to increase the impact of otherwise less severe vulnerabilities. For example, the iMessage zero-click exploit (`CVE-2021-30860`) [4] constructed and executed arbitrary boolean logic circuits built using the linguistic features of the JBIG2 image compression, to give itself the ability to execute arbitrary code via the construction of these circuits in the victim's machine. This attack was possible due to the

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112090032. and HR001119C0121. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

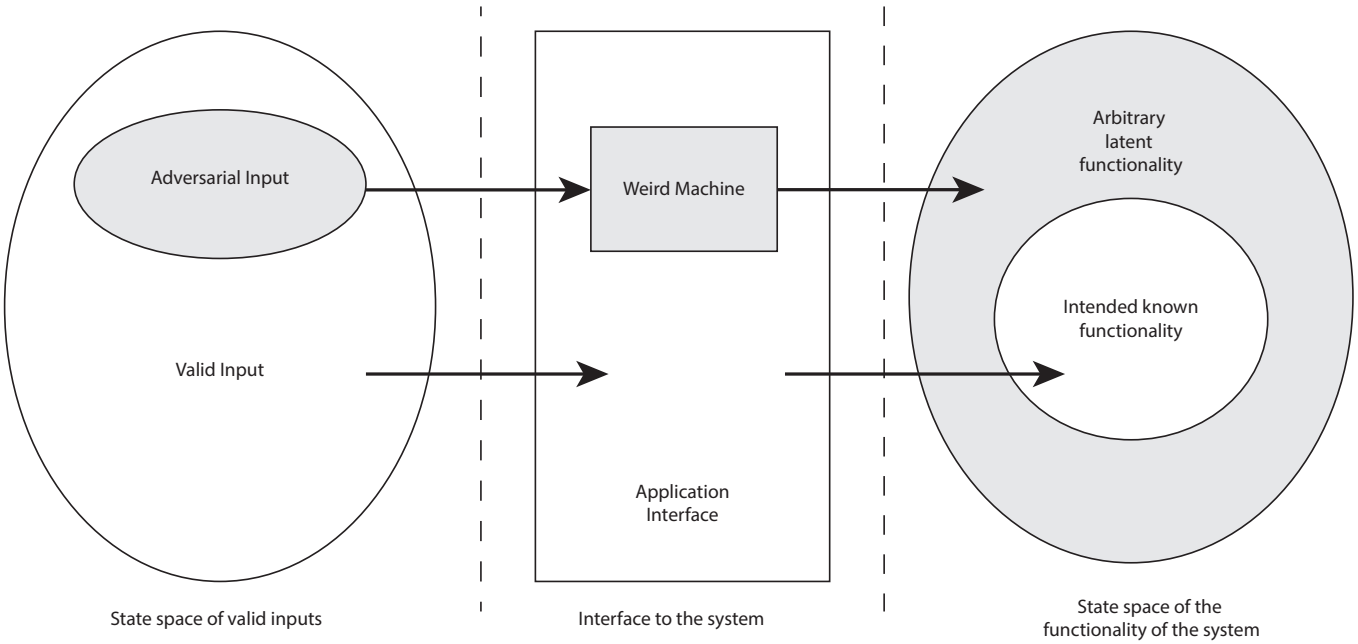

Fig. 1: This diagram illustrates weird machine vulnerabilities. We show the state space of a valid program's input language, its relationship to the application interface, and its relationship to the functionality present within a complex system. Every valid input, when consumed by the application, triggers some functionality within the complex system. Typical LangSec vulnerability families explore how crafted but invalid input can cause the victim to behave incorrectly, or how differential parsing of inputs can cause two victims to behave differently. In contrast, a weird machine provides an interface to the latent functionality already present within a complex system via a *valid* input. The box labeled weird machine shows how it is already a part of the application interface but remains unknown. A weird machine is discovered when a mapping between the latent functionality of a complex system and a set of valid inputs that trigger that functionality is made apparent.

emergence of unexpected computing artifacts present inside the victim's environment. These computing artifacts were generated from an interaction of otherwise benign linguistic features of a complex system. Similar attacks (discussed further in Section VIII-A) not only demonstrate the potential of unexpected behavior in complex systems, but also provide strong evidence for our argument that unintended computational abilities present a latent risk in complex systems which needs to be addressed for the construction of secure software systems. Therefore, to build trustworthy and secure systems it remains vital to know and clearly limit the computational possibilities expressed by a software system and their interactions.

B. Scope

Although we acknowledge the risks posed by common exploitation techniques and supply chain attacks on software systems, in this paper our research question is *only* concerned with discovering unexpected computation that presents itself inside the metadata of the RPM package management infrastructure, rather than orthogonal vectors.

This paper is, thus, not concerned with common exploitation techniques, such as those based on memory corruption or faulty logic bugs. Similarly, the paper is also not *directly* aimed at addressing supply chain attacks on package management software. The focus of this investigation lies more in discovering computational functionality that may exist within unexpected data formats — for instance, the metadata of a software package — and the challenges it poses for building secure systems.

To investigate the viability of *well-formed* metadata as a threat vector for advanced exploits, we shall limit ourselves to those scenarios where an attacker can *only* modify the metadata of RPM packages in a repository. Such scenarios may arise when a package maintainer can write arbitrary RPM metadata but does not want to tamper with the packaged application's binary or source code. A malicious maintainer may choose not to tamper with the application binary for stealth reasons, or because software-integrity checks are carried out on the application binary contained within a package on the target system. A defender looks for malicious computation inside code but expects metadata to be simple, and thus benign. Our goal is to investigate this assumption and discover programmable functionality that can be achieved by an adversary who is only allowed to manipulate the package metadata. Therefore, although the aforementioned threat model may seem impractical and overly permissive, it is deliberately

chosen to limit our scope to the computational power present in the package metadata.

Furthermore, although we mention a scenario of a malicious package maintainer, we are more concerned with what a maintainer could achieve via only manipulating package metadata than the supply-chain security issues such a scenario poses. Consequently, supply chain security risks or risks due to the uncertain provenance [5], [6] of code or repositories [7] do not lie within the scope of this work. Moreover, we will not be considering obviously Turing complete embedded scripting languages present within the RPM package manager runtime such as Lua or Bash programming languages. Rather, we are concerned with discovering the latent functionality that resides within the metadata and its interactions with the RPM package management infrastructure. Our research is primarily concerned with the ability of the RPM package manager to behave as a weird machine i.e. the ability for a package manager as a complex system to have undiscovered computing primitives within it that allow unintended computation via *well-formed and valid* input.

C. Threat Model

We assume a malicious package repository maintainer who has the ability to manipulate the metadata of packages stored on the repository. The maintainer can add, remove, or edit metadata of the packages. However, they do not have the ability to manipulate the contents of the binary data contained inside the software packages. Although perhaps contrived, this threat model is deliberately designed to ensure that our investigation remains solely focused on the computational potential of the package metadata and the RPM package management infrastructure. Further, we assume that the victim machine has this malicious repository already included in its list of repositories and that the victim updates their packages at some regular interval and installs the available upgrades (if any).

II. Application Packaging

Over the years, application packaging has become widely used in most commonly used operating systems. Application packaging is the software infrastructure that allows the distribution and installation of software on operating systems. Modern operating systems have their own application packaging infrastructure such as RPM used by Fedora, pacman by Arch, APT by Debian or Ubuntu; homebrew or macports by macOS, pkg by the FreeBSD OS, and the Windows package manager on Windows [8]. Package managers are also included in application development frameworks (e.g., [9], [10]) that are used to build apps hosted on app stores of major end-user desktop operating systems such as Windows or Mac OS.

Additionally, programming language frameworks often come bundled with their own package managers. Python's pip, Rust's cargo, Ruby's gem and Nodejs's npm are a few examples of package managers that are part of a programming language's development framework. These package managers are responsible for managing dependencies such as the set of required libraries for a software project. They also provide the added functionality to package applications developed in that framework as a software package; and the ability to install packages uploaded by others on public repositories on the host system.

Although the distribution and installation of software packages may seem like a simple task, the application packaging infrastructure comes with a surprisingly large set of features such as:

- Building software from source code
- Installing, removing, and updating software from source code or software binaries
- Querying software for detailed information and its installation status
- Verifying the integrity of the packaged software and resulting software installation on the file system
- Modifying and notifying other packages that another package has been installed
- Managing and maintaining software logs for installation, upgrade, and removal of software packages
- Managing conflicts and dependency requirements of software packages

This large set of features requires the application packaging infrastructure to be expressive. Additionally, package managers provide assurances of increased reliability and reproducibility of operations to end-users and system administrators. End-users and system administrators have, in turn, developed an increased reliance on expecting such reproducibility and reliability from the package managers. In particular, they do not expect to see differing outcomes when installing and removing an identical set of packages across identical systems but with a differing installation/removal order. Prior investigations on package management, however, have revealed such unexpected behavior [11]. Moreover, package managers have the capability to perform maintenance and upgrades of themselves and are, thus, self-modifying code [12]. This makes modeling their behavior challenging as self-administered machines can, in theory, incorporate complex feedback loops within themselves. Although they may not seem so at first glance, package managers are complex software systems.

A. RPM

The RPM package manager is used in Fedora, CentOS, OpenSUSE, and Oracle Linux operating systems. The RPM package manager packages software in 'RPM' files. These files are binary files which include a compressed archive consisting of the actual software, along with metadata. The metadata consists of information about the package such as its dependency requirements, version number, name, and signature. Figure 2 shows the layout of an RPM package file. The metadata is contained within the headers for the binary file and the archive consists of the software application files.

When the user requests a package to be installed, the RPM package manager reads the metadata from the RPM package into memory. It then uses this metadata to check if the package can be installed on the system by checking if all

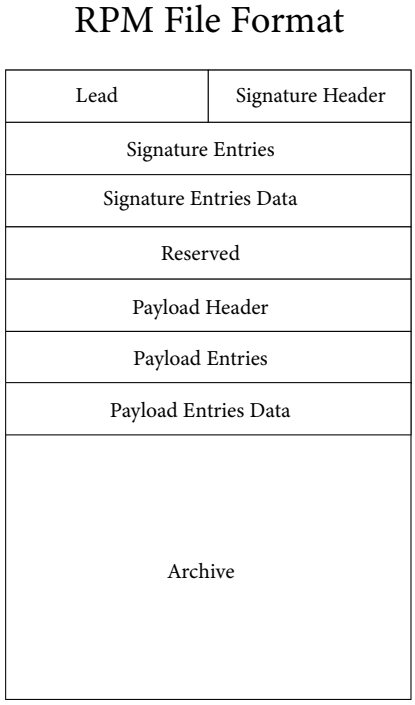

Fig. 2: Internals of an RPM package file.

dependencies are satisfied and the package does not conflict with any currently installed package on the system.

If multiple RPM files are requested to be installed, then they are all processed one after another and a transaction set is created. The user is then asked to approve the transaction set for installation. Internally, the RPM runtime starts by parsing the RPM package binary. The binary is parsed by loading it in memory and then the RPM metadata is read from its headers. If the package's dependencies are met and a conflicting package is not installed on the system, then the package manager proceeds to install the package.

B. Background on RPM

The RPM runtime consists of various components which interact together to provide the package management service. The RPM package manager's components are:

- A database to keep a record of the installed packages.
- The RPM package installer application — `rpm/yum` on the command line.
- A shell for installing, removing, and upgrading packages.
- A packaging language (RPM spec file) to define the RPM packages file.
- An embedded scripting language (RPM macro language) in the RPM spec file format.
- An event-based callback mechanism for packages to interact with each other (RPM trigger functionality).
- A dependency checker for packages.
- Package repositories on a remote server.

These various components of RPM are often complex software. For example, the Berkeley DB database is used by the RPM package manager to record package information and is stored in `/var/lib/rpm`.

C. The RPM Spec File

The package management process starts with the package maintainer who takes a software and defines its dependencies, requirements, version information and other such metadata information in a file called the RPM spec file.

This spec file is then used by the RPM build scripts to generate a ".rpm" file. Listing 1 shows an example of such a spec file. In this example, lines 7 to 20 show how dependencies and conflicts are defined inside an RPM spec file. When a RPM binary is built from the spec file, this metadata is included within its headers. The RPM dependency manager relies on this metadata for resolving package dependencies and conflicts among packages. The dependencies are of three types: very weak, weak, and strong. Strong dependencies are specified by the "Requires" and "Conflicts" clause in the spec file as shown in Listing 1. If a package is chosen for installation and its strong dependencies are not satisfied, the installation or update is aborted. On the other hand, the package manager tries to install the weak dependencies, but silently ignores them if including them in an update or installation results in a dependency conflict. The weak dependencies are specified by "Recommends" and "Supplements" clause. The "Recommends" clause, lists a forward weak dependency, which means if a package is chosen for install those listed in its metadata in the "Recommends" clause will also be included if they don't cause a dependency conflict. The "Supplements" clause, on the other hand, is a backward weak dependency which means if an package Y is selected for installation and in the metadata of another package X, there is the supplements clause listing Y, then package X will be included in the set of packages to install provided X does not cause any dependency conflicts. The very weak dependency requirements, specified by "Suggests" and "Enhances", are ignored by default. These dependencies are shown as suggestions for when a user selects a package for installation.

In addition, the spec file also provides macros, which are a limited embedded scripting language built into the spec file format to ease development for the packaging developer. The macros run when the packages are built from the spec file and allow the developers to define simple text expansions by writing code in the embedded macro language.

In addition to that, the RPM spec file consists of sections called triggers, which contain scripts that are executed when certain conditions are met [13]. Listing 1 shows an example of such a trigger that is executed when a package of name "pkg-x" having a version number less than five is installed.

III. Model of Computation

Cellular automata have long been investigated due to their ability to produce complex behaviors from simple rules. The Rule 110 one dimensional cellular automaton, in particular,

```
Name:       An Example RPM Spec file
Version:    1
Release:    1
Summary:    This package install a script that prints Hello World.
License:    None

Provides: Example

# strong dependency specifications
Requires: pkg-A
Conflicts: pkg-unzip > 2

# weak dependency specification
Recommends: pkg-B
Supplements: pkg-D

# obsoletes another package
Obsoletes: pkg-Y

%prep
# prepare source code here.

%build
# code to build software goes here.
cat > hello-world.sh <<EOF
#!/usr/bin/bash
echo Hello world
EOF

%posttrans
# code that runs at end of a transaction goes here.

%install
# install package code goes in this section.
mkdir -p %{buildroot}/usr/bin/
install -m 755 hello-world.sh %{buildroot}/usr/bin/hello-world.sh

%post
# code which runs after new package install goes here.

%triggerin -- pkg-x < 5
# trigger runs if pkg-x is less than version 5.

%files
# files installed by this package are listed in this section.
/usr/bin/hello-world.sh
```

Listing 1: This code shows an example RPM spec file. It shows the various scriptlets and trigger sections within the spec file. Comments within each section of the listing describe under what conditions the trigger/scriplet is executed. For brevity, not all scriptlets and triggers have been included.

is Turing complete [14]. We have chosen to implement this automata in the package management infrastructure to show its computational potential. This section details the formal definition of this automaton.

A. Formal Definition of the Rule 110 Cellular Automaton

The Rule 110 cellular automata is defined by a finite alphabet Σ consisting of the elements $\{0|1\}$, an infinite row of cells $\{C_i|i \in \mathbb{Z}\}$ where each cell can take a value from Σ, and a transition rule $r : \Sigma^3 \to \Sigma$ listed below:

$$
\begin{aligned}
&r(1,1,1) \to 0\\
&r(1,1,0) \to 1\\
&r(1,0,1) \to 1\\
&r(1,0,0) \to 0\\
&r(0,1,1) \to 1\\
&r(0,1,0) \to 1\\
&r(0,0,1) \to 1\\
&r(0,0,0) \to 0
\end{aligned}
$$

A configuration of the Rule 110 cellular automata is the function $F : \mathbb{Z} \to \Sigma$. It specifies what symbol is contained in each cell at index i. If an automaton is in configuration F, its next configuration F' can be obtained by applying the transition rule r to every cell $C_i \in \mathbb{Z}$ as shown below:

$$F'_i = r(F_{i-1}, F_i, F_{i+1})$$

At each discrete time step, the automata updates the values of each cell by applying the transition rule to the cell and its two neighbors.

Since the Rule 110 cellular automata is Turing Complete, for an arbitrary Turing machine M with an arbitrary input (finite tape configuration) w, it is possible to construct an initial configuration w' (with a finite number of steps from the leftmost 1 to the rightmost 1) such that the Rule 110 machine, acting on w', is equivalent to M acting on w.

IV. Implementation

We constructed the Rule 110 automaton by encoding the transition rules in the dependency requirements of RPM packages and their package version numbers. The configuration state of the automata is represented by a set of packages installed on the victim's machine. For the purposes of this section, we represent a cell at index *i* at step *s* of the automaton by a package named:

$$cell\_s\_i$$

At step k of our RPM implementation of the automaton, a cell is considered to contain "1" if and only if a package representing that cell is installed on the system otherwise we assume that the cell contains "0".

The rules of the Rule 110 automaton are encoded within the dependency requirements contained in the metadata of each package representing a cell. An RPM spec file showing this metadata information is illustrated in Listing 2.

The dependency requirements consist of two types: one is a supplements requirement and another, is a conflicts requirement. The supplements requirement instructs the package manager to include the current package if any of the packages mentioned in the supplements clause are updated or installed. However, since the supplements clause is a weak dependency, if including the package causes a dependency conflict then the supplements requirements silently ignores it. This allows us to run a check on packages representing cells that had at least one live (that is, nonzero) neighbor in the previous step. Further, to ensure only those packages that satisfy the transition rules are installed in the next step, we additionally added the conflict dependency requirements. The conflict requirements enforce that a package is not installed if any of the three rules that cause a cell to die in the next configuration are satisfied. This ensures that a package is only installed in the next configuration if and only if it satisfies the transition rules.

Initially, the victim client gets set up with a sequence of packages $cell\_0\_i$ corresponding to the initial state of the Rule 110 machine. The repository remembers the step s of the simulation (initially 0) and the indices of the leftmost l_s and rightmost r_s non-zero cells.

Each time the victim client tells its package manager to install updates, it carries out a step of the Rule 110 machine. (We observe that many clients are configured to automatically update at regular intervals.)

At step s with indices l_s and r_s, the repository constructs packages $cell\_s\_j$ for $l_s - 1 \leq j \leq r_s + 1$, following our schema. The repository also increments the version number of packages in $s - 1$.

The client's RPM dependency manager fetches the packages and computes which ones satisfy the dependency requirements and selects this subset for installation. As the RPM package manager carries out package installations, the weird machine transitions from one configuration state to another and carrying out computation. The repository updates s at each step (and can keep the limits l_s and r_s safe by decrements/increments).

It is important to note that there are no limitations on the number of packages that can be installed on a system. Therefore, the size of the set of packages encoding the configuration of the automaton is also unbounded. As a result, it is possible to encode and simulate the Rule 110 automaton with any initial configuration — and thus any Turing machine — within the RPM infrastructure.

V. Evaluation Results

In order to experimentally evaluate the practicality of carrying out computation by package installations, we carried out multiple experiments. In our first, we simulated the Rule 110 automaton and tested for correctness. Our second experiment tested the practicality of the computation mechanism by experimentally evaluating if installing a large number of packages was feasible and did not result in system instability or the package manager throwing some other kind of error. Our last experiment measured the time it took to simulate n steps of the Rule 110 automaton with non-empty inputs.

A. Experiment 1

To experimentally evaluate our hypothesis, that it was possible to use the metadata to simulate the Rule 110 automaton

```
Name:        cell_1_0
Version:     1
Release:     1
Summary:     This package represents a cell in the cellular automata.
License:     None

# If neighbors are alive, then try to install this package.
Supplements: cell_0_0
Supplements: cell_0_1
Supplements: cell_0_-1

# These dependency requirements define the transition rule for the cell
Conflicts: (cell_0_-1 and cell_0_0 and cell_0_1)
Conflicts: (cell_0_-1 unless (cell_0_0 or cell_0_1)) or (True unless (cell_0_0 or
 ↪  cell_0_-1 or cell_0_1))
```

Listing 2: This RPM spec file shows the metadata contained in an RPM package representing one particular cell of the Rule 110 cellular automata. The configuration of the Rule 110 cellular automaton is encoded as a set of installed packages in the system. Each RPM package representing a cell has the rules of the Rule 110 Cellular automata encoded in it as dependency requirements. The index of the cell is encoded in the package name. The dependencies requirements are designed to ensure that a package representing a cell will be installed if and only if the cell at that index should have the value "1". "True" is a package that is preinstalled on the system.

by installing empty RPM packages with crafted metadata, we created and set up an attacker controlled RPM repository and configured the victim RPM client to use that RPM repository as its source.

Further, the client was set up to regularly update its packages at a regular interval. The malicious repository, was then updated at regular intervals with an updated set of packages. We observed that the expected set of packages got installed when the client made its periodic call for package upgrades. As we expected, the dependency requirements ensured only the valid states of the automaton got installed on the system.

a) Experiment 2: Next, in order to test our second hypothesis that installing a large number of packages will not make the system unstable and is a possible way to carry out computation, we experimentally evaluated simulating the Rule 110 automaton with a non-empty initial state. All non-empty Rule 110 patterns expand rapidly and our goal was to experimentally evaluate if installing a large number of packages could be handled by the package manager without throwing any errors or the system becoming unstable.

We simulated the Rule 110 automaton—and to ensure a large number of packages were accumulated on the system, we made sure to not remove those packages that were no longer needed to compute the next step of the automaton. We ran our experiments overnight and accumulated more than 25000 installed packages on the test operating system without encountering any system instability.

b) Experiment 3: Our third experiment was to quantify how long it took to simulate n steps of the Rule 110 automaton. We measured the time it took to reach the nth step of the Rule 110 automaton by taking the sum of the time it took to install packages representing each step along the way. The Rule 110 was initialized with a non empty state and allowed to expand without any limitations on memory. The results of our measurements are shown in the Figure 3. We found that evaluating the next state is fairly quick (in the order of minutes) even as the automaton expands rapidly.

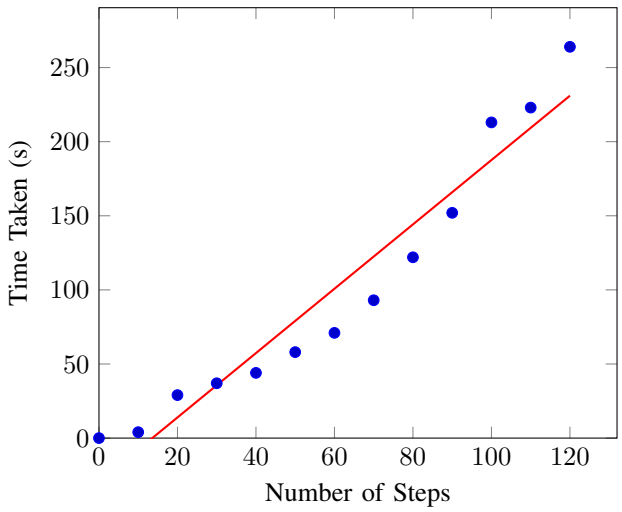

Fig. 3: This plot shows the time it took to simulate n steps of the Rule 110 automaton on the test machine.

c) Experimental Setup: We carried out these experiments on a virtual machine (with access to 1 GB of RAM, and 1 vCPU) running a CentOS 8.3 operating system with RPM version 4.14 on an laptop computer equipped with 16GB of memory and an Intel Core i7-7500U CPU.

d) Alternative construction: While investigating methods to loop, we also investigated the RPM trigger mechanism. The

RPM triggers, are programs — typically bash scripts — that are run when the trigger conditions are satisfied. Allowing the ability to embed a code snipped in the metadata, however, opens up possibilities for unexpected complexity. For instance, a package could install another packages via running a second package install command via a background process. From our experiments, we discovered that is another way to loop in the weird machine.

VI. Analytical Method

This section describes the analytical method for discovering a latent functionality within a complex system.

A. Documentation

Unexpected computational behavior is made possible due to the existence of computing primitives that already exist within a complex system. Discovering these computing primitives is challenging. To discover them, a rigorous understanding of the operational semantics of the components of the system under consideration is, therefore, necessary.

This task is extremely challenging for systems lacking in documentation — which is often the case with large complex systems. But even for well-documented systems, one requires additional documentation to break down its layers of abstractions to fully understand how it operates. Documenting the assumptions held by the constituent components of a system, and the contracts assumed among them help provide a deeper understanding of the behavior of the constituent components of the system. Often the assumptions made by such components are left unstated in the official documentation. Sometimes, the contracts between the components are assumed to always hold true, or very enforced loosely. Consequently, a sound methodology for discovering latent functionality requires thoroughly documenting the various components of a complex system, and their interaction with each other.

Understanding a complex system's behavior requires an in-depth understanding of the operational semantics of each of its components. Thus, the goal of documenting a complex system is to describe in detail the various inputs, outputs, and any effects (including side effects) a component may cause. The purpose of such documentation is to gain analytical depth in understanding the operational semantics of the constituent components of a system.

In addition, the interaction of these components with each other, and the contracts these components assume each other to hold, needs to be documented. Attention needs to be paid to the data that is sent between components. If the format of the data is expressive enough, it may allow information to be encoded in it. All possible ways a component can interact with another component should be documented, including any communication that may occur between a component due to the observation of side effects of an action performed by another component. Diagrams illustrating these relationships should be drawn out for ease of understanding. Additionally, known bugs should be looked at and analyzed as they indicate oversights in system design.

For RPM, we began our analysis by first identifying the independent components responsible for handling the package building and installation process namely, the yum dependency resolver, the RPM binary package file parser (to understand how the package metadata is encoded in the binary package), and the package builder (to understand how the spec file is converted into a binary package). Next, we identified the inputs, outputs, and any side effects for each of these components. For the RPM package builder, the input was the RPM spec file and the output was the generated binary RPM package. The generated binary package file was the input for the RPM binary file parser which reads the encoded metadata information from it and uses it to determine if a package can be installed and, lastly, the yum dependency resolver which reads the local repository cache, the local database of installed packages and the binary file of the package to install and determines which other packages need to be installed to satisfy the dependency requirements of the package.

To gain a better understanding of the function and interaction of the identified components, we read the source code of each component. Doing so allowed us to understand how the dependency information written in a spec file gets encoded into the binary RPM package and how this encoded information is interpreted by the dependency resolver. Further, by carefully reading the available documentation, we took note of the features of the RPM spec file — such as the various kinds of dependencies (weak/strong and forwards/backwards), the macro language of the RPM spec file, and the RPM triggers. Documenting these features allowed us to gain a better understanding of how a user can express complex dependency information in an RPM package. In addition, we documented the side effects that occur when a package is installed, such as file writes to the local file system, updates to the database of installed packages, and the conditions that activate triggers that are active in a system.

Documenting these features not only helps in gaining a deeper understating of inner workings of the complex system, but also provides a concise and structured reference for the exploration and discovery phase which follows afterward.

B. Exploration

The discovery of latent functionality is a discovery of the operational semantics of the constituent components of a complex system. It is, therefore, important that before the exploration phase is started the collected documentation is thorough and covers all areas within the purview of discovering unexpected system behavior.

At this stage, the assumptions held within a component and across components that were documented earlier need to be ascertained experimentally to ensure that they actually hold in the system. Any discrepancies between what is discovered experimentally and what was noted earlier needs to be added to the documentation for future reference. Afterward, each component needs to be looked at individually and analyzed for computing potential i.e. what are the possible ways it can transform the given input to its output. Crafted data

should be sent to that component to experiment and test out its computing potential. Special attention should be paid to components which have a looping mechanism within them. If there are any looping mechanisms in a component, such as a scheduler, or an update mechanism, they should be experimented with to see if it's possible to give an input that can make the loop's action dependent on the input. Such an input could be a faulting instruction, an input causing infinite recursion, or an input causing the loop action to fail and try again repeatedly. In addition, experiments should be carried out to test the computational power of the actions the loop repeats. It may be possible to carry out computation by only modifying the data given as input to the actions the loop performs rather than trying to control the loop itself. Lastly, similar experiments need to be carried out for intra-component interactions. Components, along with those parts of the system which are a result of intra-component interaction and have the ability carry out any computation, need to be identified and documented via experimentation. In particular, we are looking for primitives in the system that can provide: conditional behavior, memory, and a mechanism to loop. These parts constitute the computing primitives which, when composed together, cause unexpected behavior to arise within a complex system.

For RPM, this process involved generating packages with custom metadata via a spec file and observing if the dependency logic could be manipulated to embed a control logic of our choice. To test this hypothesis our, we first established simple logic gates by embedding the control logic of the gates in the dependency requirements. The inputs to these logic gates were a pair of the packages specified in the dependency requirements, and the package representing the logic gate was installed if the gate's output was positive. The set of installed packages on the client machine functioned as the memory of the weird machine, and the dependency requirements as the conditional behavior.

Further, we tried to combine the RPM macro language with the RPM triggers to investigate if they could be paired with the logic gates to generate loops. The macro language, however, is only evaluated at build time and was thus limited in its ability to be paired with triggers. We were thus unable to combine the macro language with the other computing primitives. However, we noted that using the RPM triggers it was possible to launch a background process to run a second package installation command when the trigger conditions were satisfied. We noted that automated package upgrades are fairly common in practice and such a scheduled update mechanism could also function as a loop for the weird machine.

C. Primitive Selection

Once the primitives have been identified and documented, experiments need to be carried out to see if multiple primitives can be composed together to perform computation. Combining primitives is fraught with challenges, as such composition is not something the system designers consider during system design and can lead to instability in the system at hand. For example, we considered the possibility of combining the RPM macro language with RPM triggers so they could activate on conditions generated by the macros. This was, however, not possible as RPM macros only run at build time. Thorough experimentation should, therefore, be conducted and all combinations of primitives should be tried out.

Attention should be paid to identifying the model of computation, the location of the internal "memory" of the model, the parts of the system which functions as the control logic of the model, the data which functions as the input to it, and to the output expressed by it. Alternative models of computation should also be kept in mind.

For instance, in one experiment we tested if RPM macros could function as string rewriting systems; in another, we tested if RPM triggers, combined with the set of packages installed on the system, could function as a neural network by encoding the activation functions in the trigger condition. Although different models of computation are equivalent, it gets easier to discover latent functionality by being aware of various ways computation can be carried out.

In our case, we found implementing boolean logic circuits the most intuitive. After we succeeded in building simple logic gates, we tried to build more complicated circuits by combining the logic gates sequentially as a sequence of package installation instructions. We noted that a package installed in the system is remembered by the system this could function as a memory for the weird machine. The dependency requirements functioned as the control logic and the package version information that is stored inside a system operated as the data of the weird machine. After identifying and combining the various logic gates into arbitrary circuits, we were able to implement the Rule 110 automaton.

VII. Discussion

Our results demonstrate the presence of latent functionality in the RPM package management infrastructure. They demonstrate that it is possible to carry out computation by only manipulating the metadata of RPM packages.

To our knowledge, this work presents the first methodology for detecting latent functionality to aid system designers and maintainers of large complex systems by preemptively discovering latent functionality within their systems. We not only provide a general approach for searching for such latent functionality within their systems; but also provide a well-documented case-study of a complex system to aid their discovery process. Having a well-documented methodology is the first major step towards building automated techniques and our hope is that by documenting a detailed methodology alongside the discovered automaton, we pave the way for future research to build them.

Whereas prior research (e.g. [2], [15]) has shown that metadata such as those of the ELF binary format and of the IA32 page table handler was computationally powerful enough to program weird machines, the computations of those weird machines were largely restricted to a single system. Our results demonstrate that weird machines are possible in networked

systems with multiple machine components (e.g., here, the victim machine and an adversarial RPM sever). The existence of networked weird machines also suggests future work is needed to discover them in cloud environments. Further, we show it is not only possible to carry out arbitrary computation by manipulating the metadata RPM package manager, but also that an adversary can also exfiltrate the results of the computation on the victim machine via encoded package dependency requests. This is possible because whenever the package manager installs a set of packages it requests those packages from the malicious server. The server is, thus, aware of the state of the Rule 110 automaton as it computes. We also demonstrate that empty RPM packages are a threat and injecting code into the supply chain of software systems is possible without modifying the binary payload of packages. These results suggest a need for a through analysis of a complex system at design time to detect the existence of weird machines. In contrast to prior work, in order to assist the aforementioned analysis, we also provide an analytical method for uncovering the underlying operating semantics of the weird machine to assist this goal.

VIII. Related Work

A. Weird Machines

Computational capabilities that unexpectedly arise in software systems are a long-time favorite of hackers and computer scientists alike, who have constructed Turing machines from components of the most unexpected software systems. An example of such a feat is the construction of a Turing machine utilizing the ELF file format's metadata [2]. Such unexpected computational abilities have been characterized as weird machines in prior literature (e.g. [1], [16], [17]) and have been discovered in various software systems. For example, unexpected computational abilities have been discovered in places as varied as the memory de-duplication mechanism of the Windows 10 operating system, Microsoft's PowerPoint, and the BGP protocol [15], [18], [19].

Weird machines is an area of research that aims to formalize an exploit from the perspective of computer science's formal language theory [17]. An exploit, seen from this perspective, can be viewed as a program utilizing properties of an existing system in unexpected ways to carry out computation. An exploit is thus constructive proof that a computation that was thought to be impossible is possible in a given system.

Advanced exploits have also used the weird machine construction to access unconstrained computational abilities and have increased their severity tremendously as a result (e.g., [3], [4], [20]). Consider for example the recent iMessage zero-click exploit (`CVE-2021-30860`) [4] which works by constructing such a weird machine to achieve its goals. It works by discovering a way — albeit by utilizing an integer overflow vulnerability — to construct and execute arbitrary boolean logic circuits, thereby giving it the ability to execute arbitrary code in the form of boolean circuits. It is important to note that although this particular exploit utilizes an integer overflow to achieve its goals, it is the construction of boolean circuits that allows it to carry out arbitrary computation. This vulnerability could have been less powerful, had the weird machine construction not been possible. Without the ability to compute via the weird machine the adversary would not have been able to calculate an arbitrary memory offset which they can overwrite thereby considerably limiting the severity of the vulnerability.

Researchers have applied weird machine formalism with success to discover and formalize exploitability in a variety of scenarios (e.g., [2], [15], [16], [21–23]). Our research adds to this growing literature of weird machines by demonstrating the computational potential of the metadata of the RPM package management infrastructure.

B. Software Supply Chain Attacks

Another orthogonal family, software supply chain attacks tries to insert malicious code into software by compromising the supply chain of the software. Such an attack could occur, for example, by compromising a library on which the software is dependent on or by compromising a server in the trusted distribution channel.

Whereas, in prior years, due to the large number of rootkit-based malware, the attention of software supply-chain research was directed more towards ensuring boot loader security [24], [25], recent incidents [26], [27] have led to an increased concern about the software supply of ordinary software as well. Supply-chain attacks on commodity software date back to at least, 2008, when attackers breached servers hosting Red Hat Enterprise Linux's package repositories and tampered with the hosted packages [28], [29]. Since, then, attackers have developed more novel, easier-to-execute supply-chain attack techniques. For instance, in 2016, Python repositories were victims of a novel supply-chain attack; this time the attack was carried out via typosquatting [30], which is an attack technique where malicious packages are uploaded with similar names to widely used packages in hopes that they will be used by unsuspecting users [31]. A similar typosquatting supply-chain attack was discovered in 2018 when malicious python libraries were found stealing GPG and SSH keys [32].

Although prior work in software supply chain security (e.g. [33]) also analyzes package metadata to look for malicious behavior, they do not consider the possibility of weird machines present in them. An analysis that looks for signals such as the presence of install scripts, comparing payloads with known malware, maintainer accounts with expired email domains, and/or packages with inactive maintainers would fail to detect a weird machine because the operating semantics of the weird machine are unknown to everyone except the attacker. Discovering weird machines is challenging precisely because of their hidden semantics. It is only after a weird machine is discovered, and its operating semantics known, that prior proposed techniques would be able to detect it. For instance, in the case of our RPM weird machine, without knowing the operating semantics of the weird machine, it becomes impossible to distinguish the weird machine from a valid set of package dependencies. Therefore, the proposed techniques

in prior supply chain security literature are orthogonal to the problem we address which is to discover the operating semantics of the weird machine in the first place.

C. Orthogonal Exploitation Techniques

Naturally, much prior work explores techniques orthogonal to weird machines. Memory corruption vulnerabilities are one of the most common ways software systems are exploited today [34]. The first known memory-corruption exploit was documented by AlephOne [35] which explained how a stack-based memory corruption could be used to alter the control flow of the target program. Soon after, heap-based buffers were exploited to redirect the control flow of a software program. Perhaps the most famous example of this would be the Netscape browser's JPEG vulnerability [36]. Other additions to the exploitation arsenal were made by the discovery of alternative ways to overwrite control data residing in memory via format-string based exploits [37] or SEH overwrite based exploits [38].

ACKNOWLEDGMENT

We would like to thank the shepherd and the anonymous reviewers for their valuable feedback.

REFERENCES

[1] T. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 391–403, 2017.

[2] R. Shapiro, S. Bratus, and S. W. Smith, "{"Weird}{Machines"} in {ELF}: A spotlight on the underappreciated metadata," in *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.

[3] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: Stealing more pie without touching the sill," *Journal of Computer Security*, vol. 22, no. 4, pp. 567–599, Apr. 2014.

[4] mitre, "CVE-2021-30860," https : / / cve.mitre.org / cgi - bin / cvename.cgi?name=CVE-2021-30860, 2020.

[5] A. Benameur, N. S. Evans, and M. C. Elder, "{MINESTRONE}: Testing the {SOUP}," in *6th Workshop on Cyber Security Experimentation and Test (CSET 13)*, 2013.

[6] W. K. Sze and R. Sekar, "Provenance-based integrity protection for windows," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 211–220.

[7] X. Liao, S. Alrwais, K. Yuan, L. Xing, X. Wang, S. Hao, and R. Beyah, "Lurking malice in the cloud: Understanding and detecting cloud repository as a malicious service," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1541–1552.

[8] D. Nelon, "Windows Package Manager 1.1," https : / / devblogs.microsoft.com/commandline/windows-package-manager-1-1/, 2020.

[9] A. Inc, "Swift Package Manager," https://www.swift.org/package-manager/, 2020.

[10] Microsoft, "How Visual Studio generates an app package manifest," https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/generate-package-manifest, 2020.

[11] J. Hart and J. D'Amelia, "An analysis of rpm validation drift." in *LISA*, vol. 2, 2002, pp. 155–166.

[12] S. Traugott and L. Brown, "Why order matters: Turing equivalence in automated systems administration." in *LISA*, 2002, pp. 99–120.

[13] rpm.org, "Trigger scriptlets," http://ftp.rpm.org/api/4.4.2.2/triggers.html, 2020.

[14] M. Cook, "Universality in elementary cellular automata," in *Complex Systems*, vol. 15, 2004, pp. pp.1–40.

[15] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, "The {Page-Fault} weird machine: Lessons in instruction-less computation," in *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.

[16] P. Anantharaman, V. Kothari, J. P. Brady, I. R. Jenkins, S. Ali, M. C. Millian, R. Koppel, J. Blythe, S. Bratus, and S. W. Smith, "Mismorphism: The heart of the weird machine," in *Cambridge International Workshop on Security Protocols*. Springer, 2019, pp. 113–124.

[17] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to weird machines and theory of computation," *USENIX; login*, vol. 36, no. 6, pp. 13–21, 2011.

[18] A. Zwinkau, "Accidentally Turing-Complete," https://beza1e1.tuxen.de/articles/accidentally_turing_complete.html, 2020.

[19] M. Chiesa, L. Cittadini, G. Di Battista, L. Vanbever, and S. Vissicchio, "Using routers to build logic circuits: How powerful is bgp?" in *2013 21st IEEE International Conference on Network Protocols (ICNP)*. IEEE, 2013, pp. 1–10.

[20] J. Wampler, I. Martiny, and E. Wustrow, "Exspectre: Hiding malware in speculative execution." in *NDSS*, 2019.

[21] J. Vanegue, "The weird machines in proof-carrying code," in *2014 IEEE Security and Privacy Workshops*. IEEE, 2014, pp. 209–213.

[22] D. Evtyushkin, T. Benjamin, J. Elwell, J. A. Eitel, A. Sapello, and A. Ghosh, "Computing with time: Microarchitectural weird machines," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 758–772.

[23] P.-L. Wang, F. Brown, and R. S. Wahby, "The ghost is the machine: Weird machines in transient execution," in *2023 IEEE Security and Privacy Workshops (SPW)*, 2023, pp. 264–272.

[24] B. Kauer, "Oslo: Improving the security of trusted computing." in *USENIX Security Symposium*, vol. 24, 2007, p. 173.

[25] R. Wojtczuk and C. Kallenberg, "Attacking uefi boot script," in *31st Chaos Communication Congress (31C3)*, 2014.

[26] S. Ikeda, "In Act of Hacktivism Open Source Project Maintainer Uses Code to Wipe Russian and Belarusian Computers," https://www.cpomagazine.com/cyber-security/in-act-of-hacktivism-open-source-project-maintainer-uses-code-to-wipe-russian-and-belarusian-computers/, 2020.

[27] Gentoo, "Project Infrastructure Incident reports," https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_reports/2018-06-28_Github, 2020.

[28] E. Mills, "Red Hat Fedora servers compromised," https://www.cnet.com/news/privacy/red-hat-fedora-servers-compromised/, 2020.

[29] redhat, "OpenSSH blacklist script," https://www.redhat.com/security/data/openssh-blacklist.html, 2020.

[30] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2020, pp. 509–514.

[31] G. Dan, "Devs Unknowingly Use Malicious Modules Snuck into Official Python Repository," https://www.redhat.com/security/data/openssh-blacklist.html, 2020.

[32] C. Cimpanu, "Two malicious Python libraries caught stealing SSH and GPG keys," https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/, 2020.

[33] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 331–340. [Online]. Available: https://doi.org/10.1145/3510457.3513044

[34] CWE, "CWE Top 25 - 2021," https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html#cwe_top_25, 2020.

[35] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[36] S. Designer, "JPEG COM Marker Processing Vulnerability," https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability, 2020.

[37] A. Sotirov, "Bypassing memory protections: The future of exploitation," in *USENIX Security*, 2009.

[38] M. Miller, "A brief history of exploitation techniques and mitigations on windows," 2007.