Needles in a Haystack: Using PORT to Catch Bad Behaviors within Application Recordings

Preston Moore[1], Thomas Wies[1], Marc Waldman[2], Phyllis Frankl[1] and Justin Cappos[1]
[1]*New York University, U.S.A.*
[2]*Manhattan College, U.S.A.*

Keywords: Domain Specific Languages, Event Processing, Environmental Bugs.

Abstract: Earlier work has proven that information extracted from recordings of an application's activity can be tremendously valuable. However, given the many requests that pass between applications and external entities, it has been difficult to isolate the handful of patterns that indicate the potential for failure. In this paper, we propose a method that harnesses proven event processing techniques to find those problematic patterns. The key addition is PORT, a new domain specific language which, when combined with its event stream recognition and transformation engine, enables users to extract patterns in system call recordings and other streams, and then rewrite input activity on the fly. The former task can spot activity that indicates a bug, while the latter produces a modified stream for use in more active testing. We evaluated PORT's capabilities in several ways, starting with recreating the mutators and checkers utilized by an earlier work called SEA to modify and replay the results of system calls. Our re-implementations achieved the same efficacy using fewer lines of code. We also illustrated PORT's extensibility by adding support for detecting malicious USB commands within recorded traffic.

1 INTRODUCTION

''Actions speak louder than words..." - Unknown

It is a well established principle that, in the wake of an application failure, its actions during execution can provide clues to the root cause. Such information cannot only help correct the cause of failure, but also prevent its repetition through the creation of better test methods. The challenge is how to identify and extract this data from large and detailed sources like application logs, system call traces, or application recordings. In other words, how does one accurately describe what activity is important and what you should do when you find it?

In considering this question, we drew inspiration from two sources. The first is our recent study that confirmed the value of monitoring and modifying an application's interactions with its environment (Moore et al., 2019). Using a technique known as SEA (Simulating Environmental Anomalies), the study demonstrated that when an application fails, the causal properties will be visible in the results of the system calls it made. Further, the study affirmed these results could be captured and simulated for testing against other applications. The second source was the significant amount of literature supporting the use of event processing techniques over large streams of data (Agrawal et al., 2008; Hirzel, 2012; Hirzel et al., 2013; Dayarathna and Perera, 2018). We posited that techniques currently used to identify problems in manufacturing environments, or patterns in network outages, could also be used to accurately recognize target sequences in large application activity streams.

Building upon these successes, we introduce a tool that utilizes event processing techniques to identify behaviors that may cause applications to fail. What makes this possible is PORT (**P**attern **O**bservation, **R**ecognition, and **T**ransformation), a new domain specific language we designed with the goal of describing these behaviors in a briefer and more easily understood manner than conventional languages. The descriptions can be used to search recordings of an application's actions across a variety of "activity representations." That is, it can search activities like system calls or remote procedure calls and determine if an application either executed a desired behavior or avoided an undesired one. Further, PORT can specify a set of modifications to be made if a particular activity sequence is encountered. By combining passive monitoring and active activity modification, PORT can aid in identifying bugs in a wide variety of programs that might be missed by other testing strategies.

In order to illustrate PORT's usefulness, we built

Moore, P., Wies, T., Waldman, M., Frankl, P. and Cappos, J.
Needles in a Haystack: Using PORT to Catch Bad Behaviors within Application Recordings.
DOI: 10.5220/0011142300003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 137-145
ISBN: 978-989-758-588-3; ISSN: 2184-2833

a prototype compiler for the language and carried out a three part evaluation. The first part consisted of creating more concise, readable and maintainable implementations of the anomalies described in our earlier work on the SEA technique (Moore et al., 2019). Next, we demonstrate how PORT allowed us to quickly add support for recorded streams of USB traffic enabling us to detect and simulate BADUSB-style attacks (Hak5, 2022) and device ID conflicts. And, finally, we demonstrate how quickly PORT programs can process recordings taken from real world network and compression applications.

The main contributions in this work can be summarized as follows:

- We create a new domain specific language, *PORT*, that allows for concise descriptions of patterns that may be found and transformed in an application's activity stream.
- We show how PORT allows for concise descriptions of anomalies.
- We demonstrate that PORT can be extended to other activity representations.
- We provide an open source implementation of PORT at: https://github.com/pkmoore/crashsimlang.

2 BACKGROUND AND MOTIVATION

Our decision to construct a large tool like PORT was not one made lightly. It came about after initial efforts revealed existing tools could not meet our requirements. In this section we explain how creating a distinct language, helped us expand SEA's applicability to other activity types, such as calls to library functions and remote procedure calls

2.1 Our Motivating Example

The initial impetus for this work was a desire to expand the use of our Simulating Environmental Anomalies (SEA) technique (Moore et al., 2019). This effort affirmed that problematic environmental properties, known as anomalies, are visible in communications between an application's components. We found that, once captured, these anomalies could be simulated and tested against other applications. Results of system calls made during execution were recorded, modified, replayed and the application's response was noted. Using this strategy, the authors were able to identify a number of bugs in major applications.

As a concrete example of the above consider the "Unusual File types" anomaly discussed in the SEA paper. This anomaly may be problematic when an application running under Linux attempts to open and read data from a file on disk, as Linux requires special procedures to handle these tasks. The SEA technique, allows a user to modify the return value of a `stat` call to simulate the presence of such a file. PORT simplifies the simulation by using proven event processing techniques and adding features that make the tool agnostic to the way an application's activity has been recorded.

2.2 Building a New Domain Specific Language

For it to work as we wanted it to, PORT needed to meet some specific criteria. For starters, it had to be able to identify specific patterns as they appear in a recording of application activity. Simple models, such as a deterministic finite automaton (i.e. a DFA or a FSA) enhanced to operate on complex structures do not work because they cannot match certain patterns. In addition, we needed a language that could easily capture and manipulate the internal contents of events, like argument data, pointer addresses, for reuse in subsequent operations.

One possibility is to deploy more expressive automata models, such as register automata (Kaminski and Francez, 1994) or session automata (Bollig et al., 2014) that can output modified streams. Several feature-rich event processing languages and libraries do have these capabilities, but the process is by no means straightforward. In many cases, producing such an output stream would require falling back on the fully-featured nature of a host language (e.g. Java) – a situation we hoped to avoid.

In the initial stages of PORT's development, we also evaluated several complex event processing (CEP) languages. These languages could provide some of the pattern and predicate matching primitives that we wished to incorporate. Sadly, these languages either lacked other required features or were too complex for the easy-to-use system we wanted to offer. Typically, programs for these complex event processing engines are written in the engine's build or host language, such as Java, Scala or Python that bring with them a great deal of boilerplate code. Recent studies have affirmed that excessive and complicated code patterns can harm understanding and maintainability (Gopstein et al., 2017). Further, it means that the author and future maintainers of a program must be fluent in this host language.

3 LANGUAGE OVERVIEW

The PORT language allows its users to completely describe a mutator, or a program that can recognize a particular event stream and optionally produce a modified version of that stream. We compile a given PORT programs into an enhanced transducer that can operate over complex data structures. The transducer consists of states, and a series of rules that govern when the current state should changes and also what modifications should be made to the input stream.

The transducer rules consist of logical comparisons between an event's parameter values, and the values stored in the transducer's registers or the literals specified directly in a program's code. Using event processing techniques, PORT maps activity onto a stream of *events*, that each consist of a unique identifier (e.g. the name of the function being called) and a list of parameter values (e.g. the argument and return values of the called function).

Parameter values are drawn from a set of basic data values, such as strings and numbers, as well as user-definable record types. PORT allows users to ignore parameter values that are irrelevant for the particular task at hand and to create a single abstract event from several semantically related, but different event identifiers. In this manner, the transducer can process abstract events rather than individually specifying each event.

A PORT transducer processes an input event stream as follows. Each item in an input sequence is examined and the transducer's internal state is updated accordingly. Output is then produced based on the rules described in the transducer's program. In this way, the transducer itself can be thought of as a sequence of *actions* that may or may not be executed based on the values in the input stream. The state of the transducer keeps track of the next action to be executed, as well as a valuation of a finite number of *registers* that hold data values. Whether or not an action is executed depends on a combination of the transducer state and the values of the current event. Therefore, executing an action consists of reading the next event from the input stream, which may update some of the registers, and then writing the next event in the output stream.

Figure 1 shows the grammar of PORT's core language. A program is split into two parts: the *preamble* consisting of type and event definitions, and the *body* of the program which consists of an action expression.

Preamble. An abstract event consists of a name and a record of named fields that hold data values of interest. Consider the event definition:

```
event rd {read fdesc: Number@0}
```

$$
\begin{array}{ll}
\textit{program} ::= (\textit{typedef} \mid \textit{eventdef})^* \; \textit{action} & p \in \texttt{pexp} ::= \{\textit{id}\texttt{:}\ p\ (\texttt{,}\textit{id}\texttt{:}\ p)^*\texttt{}\} \\
\textit{typedef} ::= \texttt{type}\ \textit{id}\{\textit{id}\texttt{:}\ t\texttt{@}n\ (\texttt{,}\textit{id}\texttt{:}t\texttt{@}n)^*\texttt{}\}\texttt{;} & \quad \mid \textit{regid} \mid c \\
\textit{eventdef} ::= \texttt{event}\ \textit{id}\ \textit{variant}\ (\mid \textit{variant})^*\texttt{;} & e \in \texttt{vexp} ::= \textit{regid} \mid c \mid e\texttt{+}e \mid \ldots \\
\textit{variant} ::= \{\textit{id}\ \textit{id}\texttt{:}\ t\texttt{@}n\ (\texttt{,}\textit{id}\texttt{:}\ t\texttt{@}n)^*\texttt{}\} & b \in \texttt{bexp} ::= \texttt{true} \mid \texttt{false} \\
\textit{action} ::= \textit{pattern}\ \texttt{->}\ \textit{id}\,(e) \mid \texttt{not}\ \textit{pattern} & \quad \mid e\texttt{==}e \mid p\ \texttt{and}\ p \mid \ldots \\
\quad \mid \textit{action}\texttt{;}\textit{action} \mid \textit{action}\texttt{*} & t \in \texttt{texp} ::= \texttt{String} \mid \texttt{Number} \\
\textit{pattern} ::= \textit{id}\,(p)\ \texttt{with}\ b & \quad \mid \textit{id} \mid \ldots
\end{array}
$$

Figure 1: Grammar of PORT's core language.

```
    | {recv fdesc: Number@0};
```

This definition maps the concrete events named `read` and `recv` to the abstract event `rd`. The parameter values of the concrete events are abstracted to a record consisting of a single field `fdesc` that holds a value of type `Number`. The notation `fdesc: Number@0` in each variant indicates that the value of `fdesc` is the 0th parameter of the corresponding concrete event. Note that all variants must map their parameters to the same record type. If an event definition describes an abstract event in terms of a single variant for which the concrete event name coincides with the name of the abstract event, then the former can be omitted in the variant.

The abstraction mechanism used for these parameter lists can also be applied to the values themselves. For instance, the 1st parameter of an `fstat` system call is a status buffer that consists of a list of values. *Type definitions* can be used to abstract such compound values into records. The following PORT code defines an abstract `fstat` event that tracks only the device identifier, inode number, and mode of the status buffer:

```
type SB {dev: String@0, ino: String@1,
         mode: String@2};
event fstat {fdesc: Number@0, sbuf: SB@1};
```

Body. The action expression in the body of a PORT program describes how the input event stream is transformed to the output event stream. An individual input event is matched and transformed by an atomic action of the form

$$id_1(p)\ \texttt{with}\ b\ \texttt{->}\ id_2(e)$$

This action matches the next (abstract) input event against the *pattern* $id_1(p)$ subject to the constraint b. The action is triggered if the name of the matched event is id_1 and its record satisfies the constraints imposed by p and b. The semantics of pattern matching is similar to the way expressions are pattern matched in functional programming languages. In particular, a variable occurring in a pattern refers to a register of the transducer. If the pattern matches the event, then the register is assigned to the corresponding value. For example, consider the pattern:

```
fstat({fdesc: fd2, sbuf: {dev:rdev, stino:
    rino2}})
```

When matched against the event

```
fstat({fdesc: 4, sbuf: {dev:"st_dev=makedev(0,
    4)", ino: 42, mode="S_IFCHR|0666"}})
```

the match would succeed and assign the value `4` to register `fd2`, `"st_dev=makedev(0, 4)"` to register `rdev`, and `"S_IFCHR|0666"` to register `rino2`.

The Boolean expression b is evaluated after the initial match of $id_1(p)$ succeeds. If b evaluates to `true` then the action takes effect. Otherwise, the match fails and the registers are reset to their original values before $id_1(p)$ was matched.

When an atomic action takes effect, the matched input event is consumed and an output event is appended to the output stream. This output event is described by $id_2(e)$ in the *output clause* of the action. If $id_2 = id_1$, then e can be a *partial* record expression, describing only those parts of the input record that should be modified by the action. Record fields that are not specified by e are copied from the input event to the output event. If the record types of the input and output events differ, e must describe the output record completely.

For example, consider the atomic action:

```
fstat({fdesc:rfd2, sbuf:{dev:rdev, ino:rino2}})
  with rfd2 == rfd and rdev == "st_dev=makedev
      (0, 4)" -> fstat({sbuf:{ino: rino}});
```

This action matches the `fstat` event given above, assuming the register `rfd` has value `4` before the match. Moreover, if the register `rino` has value `43` before the action is executed, then the action produces the output event:

```
fstat({fdesc:4, sbuf:{dev:"st_dev=makedev(0,4)"
     , ino: 43, mode="S_IFCHR|0666"}})
```

For convenience, we add a syntactic short-hand that allows one to more compactly express common pattern types. First, one often needs to express that the value of a matched record field is equal to the current value of a register. In the action above, the field `fdesc` of the matched `fstat` event must be equal to `rfd` for the match to succeed. This constraint can be expressed more succinctly by replacing `rfd2` with `?rfd` in the pattern of the action. This ensures that the matched value is equal to `rfd` without changing the value of `rfd`. The equality `rsd2 == rsd` can then be omitted from the **`with`** clause.

Next, the **`with`** clause can also be omitted altogether from an atomic action, in which case b defaults to `true`. Likewise, the output clause can be omitted and the matched input event can simply be copied to the output stream. Finally, if an action is replacing only the value of a field, and the action is not dependent on the old value, then the modified field value can be specified directly in the pattern using the notation `->` e. Here, the expression e determines the new value to be stored in the field.

Using this syntactic short-hand, the action given above can be expressed more compactly as:

```
fstat({fdesc: ?rfd, sbuf: {dev:"st_dev=makedev
    (0, 4)", ino: -> rino}});
```

Sequencing, Implicit Repetition, and Negation. Atomic actions can be sequenced to form compound action expressions that match and transform sequences of events, $action_1 ; action_2$. PORT simplifies the handling of unbounded event sequences by using implicit repetition semantics. If the next event in the input stream is not matched by the current atomic action in the action sequence, the event is simply copied to the output stream. The transducer moves on by attempting to match the next input event against the current atomic action.

Sometimes, it is necessary to constrain this implicit repetition by disallowing the appearance of certain events in the input stream before an event matched by the current atomic action is encountered. This can be done by *negated patterns*, which take the form **`not`** `id`(p) **`with`** b. If an event that matches the pattern `id`(p) **`with`** b is encountered before the next atomic action in the program takes effect, then the transducer aborts.

Explicit Repetition. PORT also supports explicit repetition of actions, which is indicated using a Kleene star, $action\star$, similar to standard regular expression syntax. The generated transducer accepts zero or more repetitions of the specified sequence of events. Output is only produced if a complete repetition of sequence is encountered.

4 EVALUATION

Once we had an implementation of PORT, we designed a set of experiments to evaluate its effectiveness in real world situations. Specifically, we aim to answer the following questions:

- Can PORT express the anomalies used by SEA to identify bugs?
- How easy is it to extend PORT to support activity representations other than system calls?
- What problems can be addressed by employing PORT on non-system-call activity representations?
- Can PORT process input streams in a reasonable amount of time?

```
1 event Statbuf {mode: String@2};
2 event anystat {stat sb: Statbuf@1}
3        | {lstat sb: Statbuf@1}
4        | {fstat sb: Statbuf@1};
5 anystat({sb: {mode: -> "st_mode=S_IFBLK"}});
```

Figure 2: A PORT program that identifies a `stat`, `lstat`, or `fstat` call and modifies the `ST_MODE` member of its `statbuf` output parameter to contain the value `"S_IFBLK"`. This indicates that the file being examined is a block device rather than a regular file.

4.1 Expressing SEA Anomalies

Given that this work is motivated in large part by a desire to expand the utility of the SEA technique, our first experiment aims to reproduce the anomalies described in (Moore et al., 2019). Specifically, we test PORT's ability to recreate the study's unusual file type mutator, and its cross-disk file move checkers, which were used to identify the bulk of the bugs that were found.

Creating the Unusual File type Mutator. For the first part of this experiment, we used PORT to implement an "unusual file type" mutator. As illustrated in Figure 2,this mutator takes an input trace that contains a call to either `stat()`, `fstat()`, or `lstat()` and modifies its result data structure so its `ST_MODE` member will indicate an unusual file type. As can be seen in Figure 2, this task can be expressed with only a few lines of PORT code. Lines 1 through 4 define what `stat()`, `fstat()`, and `lstat()` calls look like, and which parameter contains the result buffer. Line 6 generates an accepting state that, when entered, outputs a system call with a modified value in the return structure's `st_mode` field. The output can then be used to modify the results of a running application's system calls to complete the SEA technique.

The original implementation of this mutator in (Moore et al., 2019) consisted of 55 lines of Python code, much of it error-prone state management code. Our shorter PORT mutators offers several major advantages. First, they omit boilerplate code associated with general purpose languages because common functions are generically implemented, eliminating the need for users to do so manually. Next, each statement defines a specific state which ignore any system calls not dealt with in the PORT program. Finally, instead of relying on fragile string manipulation, PORT's operators make it easier to modify the parameters of system call.

Supporting Cross-Disk Move Checkers. In the second part of this experiment we tested whether PORT can implement the "checkers" used in SEA

```
1 event usbhid { src: String@0, dst: String@1,
2              data: String@13, transfertype:
                    String@10 };
3 num1 <- "00:00:1e:00:00:00:00:00";
4 num2 <- "00:00:1f:00:00:00:00:00";
5 src <- "2.1.1";
6 dst <- "host";
7 usbhid({src: ?src, dst: ?dst, data: ?num1})
      -> usbhid({data: ->num2});
```

Figure 3: A demonstration PORT program that matches USB activity indicating the '1' key is being pressed and transforming it to a new frame where the '2' key is being pressed.

to determine if an application can correctly move a file from one disk to another. This task is a common source of bugs in Linux applications because the Linux `rename()` system call does not support moving files from one disk to another. Moore et al. identified the steps required to correctly perform such a move by examining the source code of the "mv" command. The team then implemented a set of checkers to identify situations where an application does not complete this task correctly. In real world applications, these checkers were able to identify bugs in many popular applications and libraries that offer file movement capabilities.

We evaluated each of the four checkers listed in Moore et al.'s work and determined that PORT could implement three of them. For example, we were able to replace the 45 lines of difficult to read and maintain Python code in the "File Replaced During Copy" checker with a clearer 7 line PORT program. This exercise did expose one of PORT's shortcomings, which is it cannot currently implement the "Extended File Attributes" checker. Because PORT does not include a list data structure, it can not capture the values `getxattr()` and ensure they have all been applied with a corresponding call to `setxattr()`. Without this, an application cannot preserve a file's extended attributes and re-apply them after the move. Though we are considering such a feature for future implementation, we do not currently support it. Such an extension could hurt program clarity and make it harder to reason about mutator behavior.

4.2 Extending PORT to Other Activity Representations

A key feature of PORT is it easy to add support for new activity representations. To demonstrate this we implemented support for streams of USB activity. This format was chosen because of its reliance on numerous parties correctly implementing a standard protocol. Using PORT on streams of USB activity re-

quired implementing an an extension that lets PORT process USB frames and developing some way to capture communications between USB devices. For the latter, we used Wireshark because of its excellent traffic capture and dissection capabilities (Wireshark, 2022). For the former, implementing such a transformer was a straightforward task taking only around three and a half. Together, these two components allowed us to write PORT programs that could both identify patterns and transform streams of USB activity in minutes.

BADUSB. As one test scenario, we settled upon the recent type of USB-based attack known as BADUSB (Hak5, 2022). These attacks utilize small USB devices that resemble thumb drives, but when plugged into a computer, register themselves as human interface devices, and then rapidly send keystrokes to execute malicious commands. Our goal was to construct PORT programs to both recognize these attacks within a recording of a machine's USB traffic and transform an innocent recording into one containing the attack. The modified program could be replayed to assess if a computer's defensive measures are able detect the attacks.

The first PORT program we wrote detects a USB device attempting to bypass powershell's security policy. This is a common starting point for BADUSB attacks that seek to execute complex payloads. The program we wrote detects USB frames that contain a sequence of "scan codes" which spell out "powershell -Exec bypass." Detecting this string is critical because it explicitly disables security controls, a step that should only be taken under special circumstances. Using this program (see abbreviated version in Figure 3) we were able to detect the target sequence in streams of USB traffic recorded from a real computer using a standard USB keyboard. A more advanced example of PORT's capabilities with USB streams involves simulating a BADUSB attack. This program also identifies and transforms USB human interface device frames to yield key presses that spell out "powershell -Exec bypass," achieving our goal of transforming an innocent stream into a malicious one.

Device ID Conflicts. Our second test involved using SEA to simulate a USB device receiving an inappropriate "vendor ID" or "product ID" from its manufacturer. Incorrect identifiers can cause a device to not work correctly (wrongid, 2014). This malfunction has been problematic enough these devices must be disabled (barscanner, 2009). We wrote a PORT program that monitors a stream of USB traffic for USB device registrations and stores the first *vendorID* and *productID* field it encounters into registers. When subsequent registrations are encountered, their identifiers are rewritten using these stored values. This produces a new stream of USB activity where many devices share incorrect device identifiers that can be used after the fashion of SEA to test a system's response to such a mis-configuration.

Utility and Operation	Exec. Time	No. Syscalls
gzip compress file	0.110	17
gzip decompress file	0.107	35
rar compress file	0.112	109
rar decompress file	0.109	87
bzip decompress file	0.102	25
ncat server	0.103	43
socat server	0.108	71
http.server server	0.114	21
rsync client	0.132	274
ssh client	0.159	850
ftp client	0.160	891
scp client	0.135	490
telnet client	0.106	23
BADUSB	0.111	1116 lines
ID Conflict	0.117	18992 lines

Figure 4: Average time required to process the specified recording based on 100 executions.

4.3 PORT's Performance

Our final experiment evaluates the time required for PORT to identify specific patterns within a realistic set of test traces from eight widely used network applications and four popular compression utilities. Five of the applications are clients that operate by connecting to an appropriate service. Three of the applications were servers, and were recorded as they accepted a connection from an appropriate client. The compression utilities were recorded as they compressed or decompressed a file. These recordings were made with `strace` and then processed using a PORT program. For the network applications the program identifies the sequence of system calls that implement a client or server's request handling loop. The compression utility recordings were processed using a separate program that finds the read/write loop responsible for carrying out a compression or decompression operation[1]. Table 4 shows the average time required to complete the specified operation based on one hundred executions, as well as the number of system calls being processed in the recording. This performance evaluation was run on a laptop using a four core processor running at 3.4 ghz with 16 gigabytes of memory. Our PORT compiler comes with a script to reproduce these results with one command.

The results in Table 4 show that PORT's processing time increases in line with the total number of sys-

[1]Recordings are pre-processed to remove system calls related to executable loading and process creation.

tem calls in the recording. We anticipate that much of this processing cost is associated with initializing the Python interpreter and it is likely that PORT's performance is closely tied to disk throughput.

4.4 Threats to Validity

While we conducted this evaluation as rigorously as possible, there are a few areas where some ambiguity may exist. First, in our work with USB activity, we limited ourselves to US English keyboards. Other keyboard languages and designs may require enhancements to our transformer or programs. Additionally, our performance evaluation samples from only a handful of programs that were selected by popularity rather than at random. There may be programs that would diverge from the performance trend we report above. Future work can determine how widely this phenomena occurs and if any subsequent modifications are required.

5 RELATED WORK

One of the ultimate goals of developing PORT was to make it easier for developers to create tools capable of conducting program-level testing. To design such a language, we consulted previous work in processing sequences of events, such as system calls, RPC invocations or web-browser events. Below, we discuss some of the more significant work in these areas.

System Call Stream Processing Applications. System call based intrusion detection systems fall into two categories: misuse and anomaly detection. The former search for known patterns of application-specific system call sequences known as intrusion signatures (García-Teodoro et al., 2009), while the latter assumes that any deviations from "normally observed" system call sequences are malicious (Forrest et al., 1996).

Forrest et al. (Forrest et al., 1996) proposed an anomaly detection system that catalogs witnessed patterns within a database. An application's system call stream is monitored and any deviation triggers a predefined security policy.

Ko et al. (Ko et al., 1994) proposes converting each system call in a stream to a standard audit-policy record format that can be matched against program policy. However, the audit-policy can only be applied to one system call at a time, and does not support rules to recognize specific chains of system calls. Another alternative is Systrace (Provos, 2003), which uses an associated policy language to describe any action prescribed when a rule evaluates to true. Phoebe (Zhang et al., 2020) identifies patterns of system call failures during normal program execution to test the reliability of an application when a failure occurs. The downside is that more elaborate fault-injection tests cannot be generated from these sequences.

It is likely that the previously cited FSA-based programs can be improved by applying recent advances in inference modeling algorithms (Mariani et al., 2017; Walkinshaw et al., 2013; Emam and Miller, 2018; Beschastnikh et al., 2014). Yet, these algorithms lack the conciseness and flexibility found in PORT. PORT does not require training sets and is expressive enough to specify both frequent and "needle in the haystack" event sequences with just a few lines of code.

Event Stream Processing Languages and Algorithms. PORT can be categorized as a stream processing language, which means it is domain-specific and designed for expressing streaming applications. In this section we look at previous work in this area

Pattern matching over event streams is a paradigm that looks for possible matches against a previously defined set of rules. Collectively, these matches form a pattern. Languages written for this purpose are significantly richer than those used for regular expression matching (Agrawal et al., 2008), and typically provide automatic support for naming, type checking, filtering, aggregating, classifying and annotation of incoming events. They also provide many benefits over traditional stream-based text processing languages, such as sed (McMahon, 1979) and awk (Aho et al., 1979).

Though PORT is a stream processing language, it does not require all of the features typically included in this sort of system (Dayarathna and Perera, 2018). Rather PORT seems to fit within the special case known as complex event processing (CEP). Data items in input streams of these systems are referred to as raw events, while items in output streams are called composite (or derived) events. A CEP system uses patterns to inspect sequences of raw events and generate a composite event for each match (Hirzel et al., 2013)

MatchRegex (Hirzel, 2012) is a CEP engine for IBM's Stream Processing Language. Predicates defined on the individual events appearing in the stream can be utilized in the regular expression-based pattern matching engine. MatchRegex supports regular expression operators, such as "Kleene star" and "Kleene plus" over patterns consisting of predicates (boolean expressions).

Though these CEP systems are capable of recognizing the same stream patterns as PORT, they do not incorporate the transformation primitives required by the applications envisioned for PORT. CEP systems are meant to be used solely to recognize additional patterns. It is the combination and interplay of pattern matching and transformation primitives that distinguishes PORT from CEP systems.

6 CONCLUSION

One can gain a lot of value from analyzing an application's activity. Unfortunately, the volume of activity an application produces makes it difficult to separate out unimportant sequences. In this work, we demonstrate how our new domain specific language, PORT, offers a way to write concise, yet, expressive descriptions of application activity sequences. These descriptions can be compiled into programs that both recognize the described activity sequence and modify its contents in order to facilitate more active testing. We used this capability to recreate the successful programs from earlier work on the SEA technique and showed that SEA can be extended to other activity representations, such as recorded USB traffic.

REFERENCES

Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. (2008). Efficient pattern matching over event streams. In Wang, J. T., editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 147–160. ACM.

Aho, A. V., Kernighan, B. W., and Weinberger, P. J. (1979). Awk-a pattern scanning and processing language. *Softw. Pract. Exp.*, 9(4):267–279.

barscanner (2009). Barscanner Stopped Functioning. https://bugzilla.kernel.org/show_bug.cgi?id=13411.

Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. (2014). Inferring models of concurrent systems from logs of their behavior with csight. In Jalote, P., Briand, L. C., and van der Hoek, A., editors, *36th ICSE, Hyderabad, India - May 31 - June 07, 2014*, pages 468–479. ACM.

Bollig, B., Habermehl, P., Leucker, M., and Monmege, B. (2014). A robust class of data languages and an application to learning. *Log. Methods Comput. Sci.*, 10(4).

Dayarathna, M. and Perera, S. (2018). Recent advancements in event processing. *ACM Comput. Surv.*, 51(2):33:1–33:36.

Emam, S. S. and Miller, J. (2018). Inferring extended probabilistic finite-state automaton models from software executions. *ACM Trans. Softw. Eng. Methodol.*, 27(1):4:1–4:39.

Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for unix processes. In *1996 IEEE Symposium on Security and Privacy, May 6-8, 1996, Oakland, CA, USA*, pages 120–128. IEEE Computer Society.

García-Teodoro, P., Díaz-Verdejo, J., Maciá-Fernández, G., and Vázquez, E. (2009). Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1):18–28.

Gopstein, D., Iannacone, J., Yan, Y., DeLong, L., Zhuang, Y., Yeh, M. K.-C., and Cappos, J. (2017). Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 129–139, New York, NY, USA. Association for Computing Machinery.

Hak5 (2022). Usb rubber ducky. https://docs.hak5.org/usb-rubber-ducky-1/.

Hirzel, M. (2012). Partition and compose: parallel complex event processing. In Bry, F., Paschke, A., Eugster, P. T., Fetzer, C., and Behrend, A., editors, *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, pages 191–200. ACM.

Hirzel, M., Andrade, H., Gedik, B., Jacques-Silva, G., Khandekar, R., Kumar, V., Mendell, M. P., Nasgaard, H., Schneider, S., Soulé, R., and Wu, K. (2013). IBM streams processing language: Analyzing big data in motion. *IBM J. Res. Dev.*, 57(3/4):7.

Kaminski, M. and Francez, N. (1994). Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363.

Ko, C., Fink, G., and Levitt, K. N. (1994). Automated detection of vulnerabilities in privileged programs by execution monitoring. In *10th ACSAC 1994, 5-9 December, 1994 Orlando, FL, USA*, pages 134–144. IEEE.

Mariani, L., Pezzè, M., and Santoro, M. (2017). Gk-tail+ an efficient approach to learn software models. *IEEE Trans. Software Eng.*, 43(8):715–738.

McMahon, L. E. (1979). *SED: a Non-interactive Text Editor*. Bell Telephone Laboratories.

Moore, P., Cappos, J., Frankl, P. G., and Wies, T. (2019). Charting a course through uncertain environments: SEA uses past problems to avoid future failures. In Wolter, K., Schieferdecker, I., Gallina, B., Cukier, M., Natella, R., Ivaki, N. R., and Laranjeiro, N., editors, *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, pages 1–12. IEEE.

Provos, N. (2003). Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association.

Walkinshaw, N., Taylor, R., and Derrick, J. (2013). Inferring extended finite state machine models from software executions. In Lämmel, R., Oliveto, R., and Robbes, R., editors, *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 301–310. IEEE Computer Society.

Wireshark (2022). Wireshark.org. https://www.wireshark.org/.

wrongid (2014). wrong Vendor-Id and Product-Id. https://bugzilla.kernel.org/show_bug.cgi?id=87631.

Zhang, L., Morin, B., Baudry, B., and Monperrus, M. (2020). Realistic error injection for system calls. *CoRR*, abs/2006.04444.