Supply-Chain Vulnerability Elimination via Active Learning and Regeneration

Nikos Vasilakis
MIT, CSAIL
nikos@vasilak.is

Achilles Benetopoulos
UC Santa Cruz
abenetop@ucsc.edu

Shivam Handa
MIT, CSAIL
shivam@mit.edu

Alizee Schoen
MIT, CSAIL
amschoen@mit.edu

Jiasi Shen
MIT, CSAIL
jiasi@mit.edu

Martin C. Rinard
MIT, CSAIL
rinard@mit.edu

ABSTRACT

Software supply-chain attacks target components that are integrated into client applications. Such attacks often target widely-used components, with the attack taking place via operations (for example, file system or network accesses) that do not affect those aspects of component behavior that the client observes. We propose new active library learning and regeneration (ALR) techniques for inferring and regenerating the client-observable behavior of software components. Using increasingly sophisticated rounds of exploration, ALR generates inputs, provides these inputs to the component, and observes the resulting outputs to infer a model of the component's behavior as a program in a domain-specific language. We present Harp, an ALR system for string processing components. We apply Harp to successfully infer and regenerate string-processing components written in JavaScript and C/C++. Our results indicate that, in the majority of cases, Harp completes the regeneration in less than a minute, remains fully compatible with the original library, and delivers performance indistinguishable from the original library. We also demonstrate that Harp can eliminate vulnerabilities associated with libraries targeted in several highly visible security incidents, specifically `event-stream`, `left-pad`, and `string-compare`.

CCS CONCEPTS

• **Software and its engineering** → Dynamic analysis; *Scripting languages*; • **Security and privacy** → **Software and application security**.

KEYWORDS

Supply-chain attacks, Third-party libraries, Packages, Modules, Program inference, Program synthesis

ACM Reference Format:
Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3460120.3484736

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484736

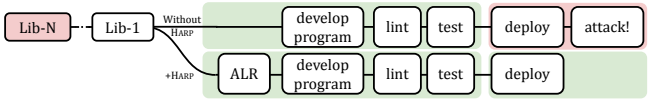


Fig. 1: Harp usage scenario. A stealthy supply-chain vulnerability can be activated long after deployment. Harp can be applied before or during development (shown) to obtain a collection of safe regenerated string libraries. Harp can also be deployed at later stages (during development or even while in production, not shown) to replace potentially malicious libraries with safe regenerated versions.

1 INTRODUCTION

Malicious adversaries increasingly employ software *supply-chain attacks* [7, 28–30, 60]. Rather than directly targeting a victim software, these attacks target a victim's supplier, exploiting the fact that the victim software depends, directly or indirectly, on software provided by the supplier. A common scenario is that the attacker purposefully inserts vulnerabilities into open source software components that are then integrated into the eventual victim software. Modern software often integrates hundreds to thousands of small components, with many components integrated not directly, but only via transitive dependencies [28, 40, 72]. It is therefore impractical for developers to audit the code that implements the integrated components—indeed, developers can easily be completely unaware of the full range of components that their system may integrate. For these reasons, even very simple, widely used components can successfully carry vulnerabilities into client software systems.

For a compromised component to remain undetected, it must typically deliver correct observable behavior to its client applications. Inserted vulnerabilities are therefore typically triggered only in very specific execution contexts and exhibit malicious behavior (such as stealthily exfiltrating sensitive data [5, 42], stealing digital assets [43, 71], or performing covert computations on the client computing platform [11, 61]) that does not interfere with correct client-observable behavior. A common scenario is that the client observes only the functional behavior of the component, *i.e.*, the results that it returns to the client when invoked, and not any malicious side effects, additional computation, or external communication that the component may perform when it executes.

Motivated by this observation, we investigate a new approach to eliminating vulnerabilities in software components. This approach

takes a potentially compromised component, explores the behavior of the component in a controlled environment to learn a model of its functional behavior (this model excludes behavior characteristic of inserted vulnerabilities), then uses the model to regenerate a new version of the component. We present a system, Harp, that applies this approach to automatically regenerate vulnerability-free versions of widely used string libraries, including libraries that operate on collections (such as lists or streams) over strings and higher-order computations that map or fold over such collections.

Deployment Scenarios: Harp supports a range of deployment scenarios. It can be used before application development starts to obtain a collection of safe regenerated string libraries that can be integrated into multiple applications developed by one or more organizations (Figure 1). It can also be deployed during development as new string libraries are integrated into the application. Finally, it can be deployed after the application is in production to replace potentially malicious libraries with safe regenerated versions.

Scope and Limitations: Our current focus is simple libraries that implement familiar utility computations with broad applicability across a wide range of applications. Such libraries comprise a compelling target for attackers because (1) they enable attackers to effectively target a broad range of computations and (2) they are often imported indirectly via higher-level libraries (as opposed to imported directly by the application developer), and as a result are typically not audited by the application's nominal developers. Indeed, many developers may easily be unaware that their applications integrate the target library.

Our approach also targets libraries whose behavior can be accurately captured with a domain-specific language (DSL). The DSL promotes effective inference and representation of the library behavior and eliminates malicious computations as inexpressible.

Our current Harp implementation targets string libraries. Such libraries implement foundational baseline functionality used widely in modern software systems. This is especially true for dynamically typed language such as JavaScript that use runtime string manipulation even for basic operations that in other languages are performed via type-safe alternatives such as type-constructor pattern matching. This is also true for many web applications, in which strings and string manipulations play a prominent role. Strings are therefore integrated, often indirectly, in the full range of JavaScript applications and are typically treated as standard components within the JavaScript ecosystem. We have developed a DSL that effectively captures the semantics of string computations and supports the efficient representation, manipulation, and inference of the underlying behavior implemented by string libraries (§4.1). Our experimental results highlight the benefits that our approach can deliver for clients of such libraries (§7).

This focused approach comes with limitations. First, it works best for widely used libraries whose computations can be captured with an efficiently inferrable DSL. We anticipate that such libraries will implement relatively simple, well understood computations. We also anticipate that the approach will work best for functional computations. Although it is possible to work with computations that perform externally visible actions such as file system or network accesses, we anticipate that it may be more difficult to ensure that the regenerated computations contain no malicious code.

Results Summary: Harp successfully eliminates vulnerabilities in 3 large-scale software supply-chain attacks by learning and regenerating the core functionality of the vulnerable library, eliminating any dependency to dangerous code (§7). We are aware of no other system that can successfully eliminate these attacks.

Applied to 17 JavaScript string-processing libraries (§7.5), Harp learns 14 libraries within a minute and all 17 under an hour. It also aborts within 5 seconds on 11 other JavaScript libraries that fall outside the string-processing domain. Harp also successfully learns and regenerates 5 C/C++ string processing modules imported as JavaScript binary modules. The regenerated libraries execute between 2% faster and 7% slower than the original JavaScript libraries and cannot use functionality beyond basic JavaScript primitives.

Key properties of Harp's synthesis algorithm guarantee that, in the limit, our proposed learning and regeneration techniques produce candidate programs with the same client-observable behavior as the original string library, if such a candidate program exists in the Harp DSL, and without malicious behaviors that fall outside client-observable behavior.

Contributions: This paper makes the following contributions:

- **Active Learning for Vulnerability Elimination:** Given a component to regenerate, Harp chooses inputs, feeds these inputs to the component, and observes the resulting outputs to infer a model of the client-observable functionality that the component implements. Harp executes the component in a controlled environment to discard any behavior that is not observable in the direct functional interactions with the Harp learning system.
- **Domain-Specific Language:** Harp builds the inferred model as a program in a DSL for capturing string computations, including computations over collections of strings and computations that `map` or `fold` over such collections. This approach provides important benefits: (1) *Tractable Learning Without Overfitting:* The DSL acts as a strong regularizer that focuses the inference on the target class of string computations. It prevents overfitting and promotes efficient inference that typically requires only automatically generated input-output observations to precisely identify a specific string computation within the larger class of string computations. (2) *Safe Modeling:* The DSL is designed to express only legitimate string computations. The inferred model therefore excludes behaviors that augment string computations with auxiliary malicious computations.
- **Regeneration:** Given a string computation in the DSL, Harp regenerates the computation in the desired target programming language, with any malicious behavior in the original component not learned during inference and discarded in the regeneration.
- **Experimental Results:** It presents results that characterize the ability of Harp to learn and regenerate a range of string libraries and highlight its ability to eliminate several software supply chain attacks that target string libraries.

2 BACKGROUND & EXAMPLE

We use the `event-stream` incident [41, 61], where a popular stream-processing library was modified to steal bitcoins from carefully selected targets, as an example of the attacks Harp is designed to eliminate. At the time of the incident, `event-stream` was used

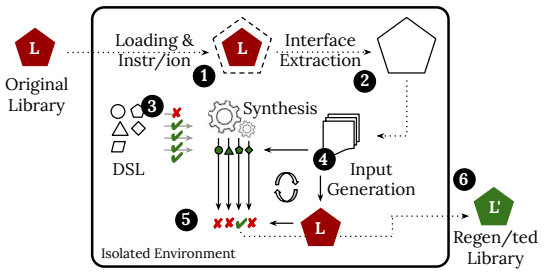


Fig. 2: Overview. In an isolated container environment, Harp loads a library and inspects its interface. Using increasingly sophisticated rounds of exploration, it generates inputs, provides these inputs to the library, and observes the resulting outputs to infer a model of the library's behavior as a program in a domain-specific language.

(imported either directly or indirectly) by thousands of applications and averaged about two million downloads per week. When its author handed off maintenance to a volunteer—common practice in open-source development projects—the new maintainer added an obfuscated, malicious library called `flatmap-stream` as a dependency to `event-stream`.

The malicious `flatmap-stream` library is designed to harvest account details from select Bitcoin wallets. If run in the dependency tree of a specific Bitcoin application called Copay, `flatmap-stream` loads Copay's `account` module containing the Bitcoin wallet credentials of the user using Copay. It then overwrites the `account`'s `getKeys` method with one that copies and stores the credentials on the side. It then loads the `http` module, and `post`s the credentials to a remote server, before returning the results to the caller method.

The desired client-observable behavior of `flatmap-stream` maps a function over a stream. Removing `flatmap-stream` therefore breaks the client. The attack succeeds by performing *effects*—loading `account`, overwriting `getKeys`, importing `http`, and calling `post`—that do not interfere with the client-observable behavior. The attack is not detectable by static analysis, because the attacker employed a series of dynamic encryption passes, nor dynamic analysis because the malicious code activates selectively far from development and testing: only when `event-stream` was part of Copay's dependency tree, only when run on the "live" bitcoin network, and only on users that had a balance of 100 bitcoin or more [52]. When run in any other context, the compromised version of `flatmap-stream` exhibits identical behavior as the correct version.

Applying Harp: Harp can directly target a specific dependency or a library that integrates multiple dependencies. The following line applies ALR directly to `flatmap-stream`:

```
harp -ft js flatmap-stream
```

Harp first loads `flatmap-stream` in an isolated container environment and applies lightweight program transformations to instrument its execution (①, Fig. 2). This instrumentation records operations such as library imports, file-system reads, and global variable accesses that `flatmap-stream` performs. Harp also extracts information about the library interface (②). This information includes the number of returned methods and fields and the number of arguments for each method. Harp then runs `flatmap-stream` on synthesized inputs, to extract information about the types of each argument.

Harp next uses `flatmap-stream` to synthesize a program in the Harp DSL as follows. It iteratively generates candidate programs in the Harp DSL, filtering out candidate programs that do not match the extracted type information (③). It then executes the original version of `flatmap-stream` and remaining candidate programs on an iteratively increasing set of generated inputs (④). It observes the parameter and return values of the original library and the candidate DSL programs (these parameter and return values are the client-observable behavior). It filters out candidate programs that exhibit different client-observable behavior than the original library (⑤).

As the Harp inference algorithm executes, it maintains a set of candidate DSL programs that exhibit identical behavior as the original library on the current set of generated inputs. If the input generation algorithm enumerates all possible inputs in the limit (many such algorithms exist for countably infinite inputs such as strings), this process will, again in the limit, produce a DSL program with the same client-observable behavior as the original library (⑥), if such a DSL program exists (§5). In practice, Harp is usually able to synthesize a unique successful candidate program within an hour and typically within minutes (§7). Harp also implements a `--quick-abort` option that immediately aborts the search if the Harp instrumentation detects any non-client-observable behavior such as file system, environment variable, or network access.

In our example, the malicious `flatmap-stream` behavior is not triggered in our isolated container environment and `flatmap-stream` exhibits fully correct behavior. Working with 2,536 inputs, Harp takes 1.4 seconds to synthesize the following correct DSL program, which exhibits identical behavior as the correct version of `flatmap-stream`:

```
f s = map (squash n) | "{(c)}"
```

Here `f` maps the function `squash n` over the elements of `s`, thereby flattening `s`, and then pipes each of the results to an output pattern, which simply outputs its input element.

Harp then compiles the synthesized DSL program to the following JavaScript library:

```
const libHarp = require('./lib-harp.js');
let program = (f, isAsync) => {
  const stream = new libHarp.Stream();
  stream.addOperation(libHarp.squash);
  stream.addUserOperation(f, isAsync);
  return stream;
};
module.exports = program;
```

The compiled regenerated library is a direct translation of the inferred Harp DSL program. It links to `lib-harp`, a module that supports Harp's core functionality (part of the TCB, §3).

3 THREAT MODEL

Harp protects against an adversary that fully controls a target component and can modify it in any way. By preserving the client-observable functionality, the adversary aims to execute undetected attacks when the component is integrated into an application. Examples of modifications include added functionality that reads from the file system, sends messages over the network, reads environment variables, or writes to global variables.

Module	m	:=	$s \mid m\ .\ s$
Statement	s	:=	add $c\ l \mid$ del $l \mid$ at $l\ b \mid$ repeat $n\ s$
		|	toggle $p\ o \mid$ map $\pi \mid$ fold $b \mid$ split $\sigma\ s\ \sigma$
Location	l	:=	$i \mid /p/$
Index	i	:=	$n \mid$ start $\mid$ end
Predicate	p	:=	not $p \mid$ or $p\ p \mid$ and $p\ p \mid k$
Pipeline	π	:=	$\pi\ .\ \pi \mid b \mid$ squash $n \mid o$
Char. Class	c	:=	$\alpha \mid n \mid \sigma \mid$ regex $r \mid$ capt k
Output	o	:=	$o\ .\ o \mid c \mid \perp \mid b\ \perp$
Capture	k	:=	$k\ .\ k \mid p \mid p\ \perp$
Built-ins	b	:=	+ | − | × | % | `Math.*` | `String.*`
Regex	r	:=	$r \mid \alpha \mid n \mid \sigma \mid {}^{*} \mid {}^{+}$
Special	σ	$\in$	Σ_1
Alpha	α	$\in$	Σ_2
Num	n	$\in$	$\mathbb{N}$

Fig. 3: The Harp library DSL. The domain-specific language (DSL) captures the space of inferrable program libraries.

For ALR to regenerate a successful replacement, the library must exhibit the correct behavior during testing and this correct behavior must conform to the ALR DSL. We anticipate that our target class of vulnerabilities will typically satisfy these two requirements—the goal is typically to provide the client with the correct functionality while either (1) stealthily opening up a remotely exploitable vulnerability or (2) silently exfiltrating data or modifying the system on which it runs. To avoid exploitation during learning, ALR runs the target library in a controlled isolated environment.

An attacker may also simply remove the library from the ecosystem, disabling any application that depends on the library. By replacing the library with a regenerated local version before the original library is removed, ALR eliminates the dependence and enables applications to continue to operate successfully even in the absence of the original library.

The language's runtime environment, bindings for locating and loading libraries, a small compiler offered by Harp and the associated `lib-harp.js` runtime-support library are all part of the trusted computing base (TCB). To capture possible interactions between libraries, we assume Harp is loaded before other libraries. We also assume that other libraries do not cooperate with the target library to attack the system.

4 ACTIVE LEARNING & REGENERATION

Harp combines three components: a DSL for specifying string-processing computations (§4.1), an algorithm for inferring computations in the DSL (§4.2), and an input generation component that produces the inputs for the inference algorithm (§4.3). The three components work in tandem, aided by lightweight runtime interposition for mapping the interface of a library (§4.4).

4.1 Domain-specific Language

Fig. 3 presents the Harp DSL. The DSL specifies the set of all programs that can be constructed by Harp, broken down into a few broad classes: (1) computational primitives, which apply transformations on their input, (2) built-in primitives for number and string manipulation commonly offered by high-level languages, (3) input ranges, over which these primitives are applied, and (4) character classes, used for pattern ranges and primitives. More complex classes often combine less complex ones.

Computational primitives: Computational primitives are either statements or pipelines. Statements include add and del primitives for introducing and deleting characters and higher-order map and fold primitives for applying a first-order primitive over a range. Pipelines apply a series of operators to a collection of elements in an input stream—optionally recursively to elements of their elements.

These primitives are Harp's primary building blocks. Fig. 4 presents the operational semantics. The transition function $\Longrightarrow$ maps a computational primitive within our DSL to its output value. For example, the primitive add accepts a character c, a location l, and a string s, and returns a string that is the result of adding the character c at location l in s. All primitives accept a string s on which to operate as their final argument. Strings are encoded as lists of characters, list concatenation is encoded as $\cdot$, and operations encoded in sans – serif are built into Harp—for example, match accepts a predicate p and a string s and returns three character lists: (1) a string s_1 up to (but not including) the match, (2) the matching string s_2, and (3) the rest of the string s_3 following s_2 in s.

Built-in primitives: This class contains primitives offered commonly by the standard libraries of different high-level programming languages, including operations for arithmetic—*e.g.*, `log`, `sqrt`, *etc.*—and string manipulation—*e.g.*, `toUpper`, `toAsciiCode`. The class of built-in primitives reimplements these operators from scratch to address two challenges. The first challenge is that different languages offer different operators under different names; the Harp DSL unifies a common subset under a common set of identifiers. The second challenge is that the invocation patterns of such primitives are different for different languages—for example, JavaScript's `n.toString` is invoked directly on a number n, whereas Python `str(n)` takes n directly as an argument. Harp DSL introduces these operators as functions whose first argument is the input string.

Input ranges: Computational primitives often take as arguments a location within the string. In their simplest form, locations are indices relative to the start of an input segment, which can be a string or a substring within that. For example, the index start in the expression (at start `String.toUpper`) matches the beginning of the string.

Locations can also be predicates that pattern-match on the form of the string. Predicates are formed by the composition of a simpler set of base predicates. Composition operators include negation, disjunction, and conjunction. Base predicates are centered around a simple pattern-matching language that includes characters, numbers, "*" (Kleene-star superscript), and "+" (Kleene-plus superscript). For example, the predicate $/a^+/$ in at $(/a^+/)$ (`String.toUpper`) matches one or more a characters.

Character Classes: The DSL includes three sets of characters. Two of these sets come pre-configured and built into the DSL: (1) the set of integer numbers and (2) the set of alphanumerics—including number characters "0" to "9", lowercase letters "a" to "z", uppercase letters "A" to "Z", and punctuation symbols. The third set contains characters that are special to a particular computation. The members of this set are discovered during the learning phase via input generation (§4.3).

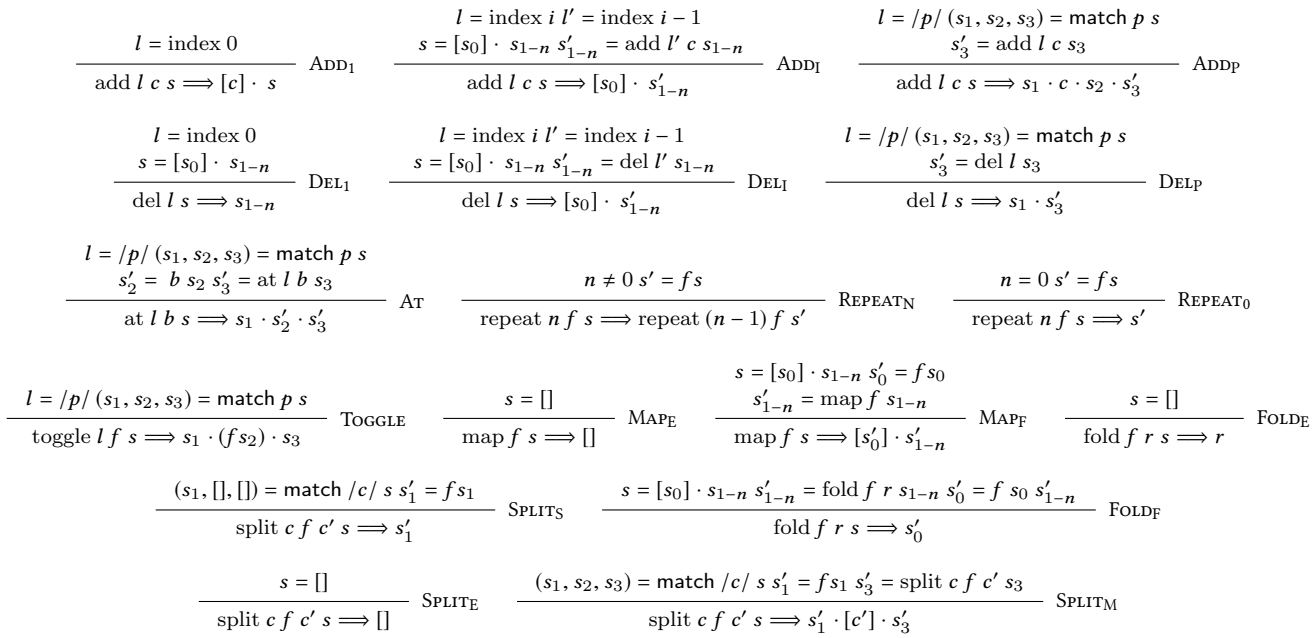


Fig. 4: DSL Semantics. A subset of HARP's DSL semantics, describing HARP's computational primitives.

Capture and output expressions: Two examples of how simple elements like character classes and built-in primitives are used to construct more powerful primitives are capture and output expressions. `toggle`'s second argument is an output expression, which can be thought of as a format string that one would pass into a function like C's `printf`, describing the formatting of the function's output. It can contain literal characters, as well as special identifiers, which are bound to strings that were matched as part of `toggle`'s first argument, its predicate, and captured using a capture expression. For instance, whenever toggle encounters any character preceding an uppercase character in the program:

```
toggle f'{/./(a)}{/[A-Z]/(b)}' '{(a)}- {
    ↪ to_lower (b)}'
```

it will output the first character it matched—which it assigned to variable a in the capture expression—followed by a dash, followed by the captured uppercase character (assigned to b) converted into lowercase.

4.2 Synthesis Algorithm

Alg. 1 outlines the HARP program inference algorithm. The algorithm takes as input a black-box reference implementation r and produces as output a DSL program with identical behavior as r on the generated inputs I. Given a reference library R, HARP invokes Alg. 1 on all functions r in R.

As Alg. 1 runs, it maintains an iteratively increasing size n of DSL programs and list of generated inputs I to consider. At each step of the algorithm it first executes *generateInputs*(I) to augment the current list of inputs I with additional generated inputs as described in the next section (§4.3). It then runs the reference implementation r on the inputs I, producing a list of input/output pairs $IO = [\langle i_1, o_1\rangle, \ldots, \langle i_k, o_k\rangle]$. These outputs are considered ground-truth outputs, because they are generated by the reference implementation r. For example, applying $run(r, I)$ to r = `length` and I = `["a", "bb", "ccc"]` produces IO = [⟨`"a"`, 1⟩, ⟨`"bb"`, 2⟩, ⟨`"ccc"`, 3⟩].

Alg. 1 next invokes *typeConstraints*(IO) to collect a set of sound type information T for the values in IO. This procedure includes several type-inference tests checking whether the values in IO represent numbers, whether their length is longer or shorter than the input length, and whether they contain any special characters. For example, the result of calling *typeConstraints*(IO) on IO = `["bb", 2]` would return *String* → *Number*.

Navigating the search space: The algorithm next prepares the search space of candidate DSL programs, which is parametric over the maximum number n of terms used in the program—*i.e.*, the size of the abstract syntax tree (AST) of each candidate DSL program. The algorithm generates the search space by invoking *allPrograms(n,T)*, which takes a number n and a set of sound type constraints T and returns a set P_n containing all of the programs of size n that satisfy the type constraints T. Consider an example where (1) all programs of AST size 1 are captured by the set of single-term programs {`count`, `toString`, `+`, `-`, `*`}, and (2) the type constraints include `Number`→`Number`→`Number`. Then P_1 = {`+`, `-`, `*`}.

The algorithm then invokes *filter*(P_n, IO) to eliminate all candidate programs in P_n whose input-output behavior does not conform to (I, O). This procedure eliminates candidate DSL programs with behavior that is not identical to r—*i.e.*, programs for which not all inputs in I produce outputs in O. As appropriate, *filter*(P_n, IO) may also generate more input-output examples to further differentiate between candidates and thus prune the search space even further. The result is a set of candidate programs P, all of which implement r's input-output behavior on the input-output examples.

Input: Reference Implementation r
Output: List of Inferred DSL programs $< d_1, \ldots, d_k >$
$n \leftarrow 0$; $I \leftarrow \emptyset$
while not done **do**
  $I \leftarrow generateInputs(I)$
  $IO \leftarrow run(r, I)$
  $T \leftarrow typeConstraints(IO)$
  $P_n \leftarrow allPrograms(n, T)$
  $P \leftarrow filter(P_n, IO)$
  $n \leftarrow n + 1$
end
return $getOpt(P)$

Algorithm 1: The Harp program inference algorithm. Given as input a black-box reference implementations r, the algorithm produces a DSL program with identical behavior as r on the generated inputs I.

Termination: In principle, Alg. 1 can run indefinitely. In practice, the algorithm maintains some additional information on the side (not shown in Alg. 1). First, the algorithm is configured to run up to a time limit—either a limit $t_{\bar{r}}$ per reference implementation r in the reference library R or a limit t_R for R overall. If only $t_{\bar{r}}$ has been specified, then t_R is calculated as $t_{\bar{r}} \times |r_{1-n}|$ spread fairly across all functions r_{1-n} in R; when Harp timeouts for one of the methods, it simply outputs `Nil` and moves to the next r_i in R. When t_R is specified, Harp can allocate this time as it sees fit (see parallelism in §6.2). The combination of the two limits is possible too, instructing Harp to spend no more than t_R minutes overall, with no more than $t_{\bar{r}}$ minutes per function r in L.

Using timeouts, Alg. 1 may need to exit the inner loop with a P equal to the empty set. If this happens, $L'.r$ is assigned `Nil` which is important for partial regeneration, in cases where only a fraction of a library's functionality has been successfully regenerated.

Finally, Alg. 1 inspects the set P. If P is not empty, it ranks the candidate programs in P by invoking and returning $getOpt(P)$, which returns the highest-performance program in P. During the $filter(P_n, IO)$ procedure, the synthesis algorithm collects information about the runtime performance of the candidate DSL programs. Some of the inputs in this phase are large, to make any differences in overhead more pronounced. This information is then used by $getOpt(P)$ to rank candidates based on their runtime performance, returning the DSL program with the best performance.

4.3 Input Generation

Harp generates inputs for each reference function r in R and executes r to obtain the input-output pairs. Harp chooses these inputs to gather a variety of output values that, combined, highlight key properties of r's behavior. As Harp does not know beforehand what input streams are the most appropriate for inferring the behavior of a black-box r, it adopts an active learning algorithm to generate the inputs. There are two kinds of inputs Harp is interested in: (1) primary inputs, which are the strings on which the string-processing computation is applied and (2) secondary inputs, which are other parameters of r affecting the specifics of the string computation. All mutations described below are applied concurrently in iterative rounds providing information or eliminating candidates. When a mutation iteration results in no candidate eliminations, this phase of input generation terminates and saves the set of candidate regenerations.

Value	v	:=	$p \mid \{s : v, \ldots\} \mid [v, \ldots]$
		|	$\lambda(x, \ldots).x \mid \lambda(x, \ldots).\mathsf{str}(x)$
Primitive	p	:=	$s \mid n \mid b \mid \bot$
Boolean	b	:=	`true` | `false`
String	s	$\in$	Σ
Number	n	$\in$	$\mathbb{N}$

Fig. 5: Harp's secondary-input DSL. This language captures the space of possible inputs to secondary arguments.

Primary inputs: The primary input of a string processing function is a string—a collection of characters—or a collection of strings. Input characters are particularly important because they may affect r's processing locations—thus Harp attempts to quickly discover a set of special characters Σ_1. Harp generates primary inputs that exercise certain properties in an attempt to understand which of their characteristics affect r's output. The key insight behind such discovery is that that string computations are generally applied over linear data structures that encode control and data characters in a single data stream. For example, consider the following string:

```
one:two three-four
```

Different processing primitives may be affected by different characters. For example, a (`to-upper`) function converting to upper-case operates on the entire string, a (`split :`) function splitting on ":" will match only the corresponding character, and a "`mask *`" function replacing characters with "*" will only match a subset of characters. These and other examples are shown below:

```
one:two three-four     upper-case
one:two three-four     split ':'
one:two three-four     mask-cc
one:two three-four     camelCase
one:two three-four     array-first
```

To discover this set, Harp generates strings with a combination of letters, numbers, and punctuation symbols. As soon as some of these inputs start affecting the results, Harp narrows down the set of symbols by mutating only parts of the input string.

Secondary inputs: Functions in the reference library R rarely accept only strings as their inputs. That is, while the processing targets the primary input string, other arguments part of the method's interface need to be provided. For example, a simple `count(s, c)` method that counts all occurrences of `c` in `s` takes two arguments. To understand the effect of other inputs to the computation, Harp introduces a small DSL describing possible secondary values (Fig. 5). To maintain acceptable performance, Harp generates only constrained inputs of these types—both in terms of size and complexity.

These values can be summarized into two broad classes. The first class is composite values such as lists, objects (maps), and functions. The DSL includes only two functions, helpful for cases when r is a higher-order function. These two functions are designed to have types that are permissive and will likely not throw exceptions. The first function simply returns its first argument, matching any fold-like operations; the second function returns its first argument as a string, covering additional use cases where the first-order function is expected to return strings—highly likely due to the domain of Harp. Both functions take a variable number of arguments so as to be compatible with any invocation in the black-box r.

The second class involves primitive values such as strings, numbers, booleans. The value $\perp$ corresponds to `null` or `undefined` values; such values are important for understanding the default parameters or behavior of a computation.

Enumerability: As Alg. 1 executes, it considers larger and larger sets of programs and inputs I, with the current set of programs P_n containing all programs of size n or less and the current set of inputs I generated by (repeated calls to) $generateInputs(I)$. The algorithm will eventually consider every program in the DSL. If $generateInputs(I)$ will eventually enumerate every possible input, then in the limit the algorithm will either (1) converge to a DSL program or programs P all of which have identical behavior as the reference library R on all inputs (if such program or programs exist in the DSL) or (2) determine that no such program exists (see Section §5). To promote fast convergence to a correct DSL program, the current Harp $generateInputs(I)$ algorithm is designed to prioritize strings that quickly disambiguate candidate computations over strings.

4.4 Mapping Library Structure

We next cover a few details on how Harp (1) regenerates constant fields, and (2) discovers the structure of a reference library R.

Constant fields: The majority of string-related functionality is expected to be exposed as functions. At times, however, R may contain fields other than functions—*e.g.*, a map of country names to dial-in prefix codes. In these cases, Harp can copy the structure into the regenerated library using runtime meta-programming facilities: it traverses R's return object to identify and copy such values directly.

In rare cases, these inputs are hidden behind a functional interface that does not allow meta-programming facilities to permeate through. In these cases, Harp resorts again to active learning—but its input generation leverages a built-in dictionary of common English words. Harp attempts these words under various combinations and capitalizations to gain more information about the mapping.

Field discovery: To apply the techniques described earlier, Harp needs to know how to interact with R and how to feed it inputs. To answer this, Harp first loads the original library, an operation that returns an object that contains the values `exported` by the library. These values may include functions or other directly accessible fields. The way Harp interacts with these fields depends on whether the functionality about to be regenerated has been explicitly named by the developer using Harp. If it has been named, Harp indexes only the named functions from the returned object. If there is no explicit naming involved, Harp uses runtime meta-programming to traverse the returned object in order to understand and regenerate the structure of the library.

5 GUARANTEES

A key correctness guarantee is that the Harp synthesis algorithm (Alg. 1) will only produce string computations whose behavior is captured by the DSL in Figure 3. Recall that Algorithm 1 maintains a current program search size n, set of input-output examples I, O obtained from executions of the original library L, and set of programs P in the Harp DSL. The Harp synthesis algorithm provides the following key correctness guarantees:

- All programs in P exhibit identical behavior as the original library L on the list of generated inputs I (the call to $pruneSpace$ in Algorithm 1 filters out all DSL programs whose behavior differs).
- The set of DSL programs P contains all DSL programs of size n or less that exhibit identical behavior as the original library L on the list of generated inputs I.

These guarantees have an immediate corollary:

- If the original library L has the same behavior on all inputs as some DSL program f' and f is of a given size n or less, then $f \in P$. Moreover, if $P = \{f\}$ (i.e., f is the only program in P), then the newly synthesized library L' has identical behavior as the original library L on all inputs.

If the $generateInputs$ function eventually enumerates every possible input, then, in the limit Alg. 1 will consider all programs in the DSL. More precisely, for any specific input and program of some size n, there is some finite execution of the algorithm that will generate that input and consider that program. This fact ensures the following guarantees:

- If the original library L has the same behavior as some DSL program f (of some size m), then at some finite point in the execution of Algorithm 1, $f \in P$ for all future execution points.
- If the original library L has different behavior than some DSL program f (of some size m), then at some finite point in the execution of Algorithm 1, $f \notin P$ for all future execution points.

These guarantees provide a form of correctness in the limit—as the algorithm runs, it (1) will eventually (in finite time) find the correct DSL implementation of the original library L if such a correct program exists in the DSL, and (2) will eventually (in finite time) filter out any DSL program whose behavior does not match the original library L on all inputs.

6 REFINEMENTS

We next present several Harp refinements.

6.1 Isolated Learning

To avoid exploitation during ALR, Harp interacts with target libraries in an isolated container environment. Harp first launches a Docker container and imports the library in the context of an TCP server. Harp then traverses the object returned by the import statement to create a remote-procedure-call (RPC) shim, which it then writes in the host file-system.

Harp's ALR scaffolding infrastructure on the host environment loads the shim module to interact with the target library. For every invoked library function, the RPC shim serializes the arguments and send them to the server executing in the Docker container. Harp invokes the corresponding function and returns the results back to the shim, which delivers them to Harp running on the host environment. The channel between the RPC client function and the corresponding function running in the container is encrypted using NaCl authenticated encryption primitives [4].

6.2 Synthesis Acceleration

Type Guidance: Harp leverages sound type information to guide its choice of DSL terms. This is achieved through a few different means, starting by checking the size and type of the output. If the

```
let _o = o, o = {};
o.f = (...args) => {
  accesses[_o.f] = true;
  return o.f(...args);
}
```

(a) Object-wrapping fragment

```
var context = {
  eval: harp.wrap(eval),
  Number: harp.wrap(Number),
  Array: harp.wrap(Array),
  //...another 138 similar lines
}
```

(b) Custom context creation

```
function (cxt) {
  let eval = cxt.eval;
  let Number = ctx.Number;
  //...another 139 similar lines
exports = (s, l, c) => s.padStart(l, c)
}
```

HARP additions

(c) Context rebinding

Fig. 6: HARP's detection of library-external side-effects. HARP's basic `wrapping` traverses objects and wraps fields with inline monitors. HARP uses this transformation to create a new name-to-value context by `wrapping` all values available in a library's top-level scope (b). The modified context is bound to the library by enclosing the module source (half-visible code fragment, in its original indentation) in a closure that redefines all non-local variable names as closure-local ones, pointing to values from the modified context.

output is significantly smaller, then a fold-like reduction is likely to play a prominent role in the regenerated computation. Additionally, if the output has a certain type—such as a number or a boolean value—then that type should featured in the first-order function used as part of the reduction. Outputs whose size is close to that of the input string often correspond to `add` or `at` constructs.

The study of more complex outputs is also possible, as HARP can leverage meta-programming available by the source language to introspect the value returned by L. This is different from other domains where active learning is applied through serialization-deserialization interfaces that encode all values as strings, and thus obscure the true types of the values returned by a program fragment. These refinements can prune the synthesis search space significantly.

Term Weights: Different (classes of) terms from HARP's DSL have different likelihoods of appearing in learned DSL programs. For example, many regenerated string-processing libraries `add` or `delete` characters. HARP uses such likelihood information to guide synthesis, by generating higher-likelihood terms in the DSL with higher probability HARP explores the space of candidate programs.

Term weights depend significantly on the types of the inputs and outputs. For example, if the output is a number then reduction statements such as `fold` and built-ins such as × and + are more likely to appear in the regenerated program.

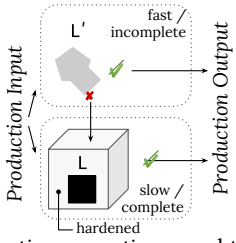
Parallel Synthesis: HARP's synthesis features ample opportunities for parallelization. One opportunity occurs in candidate generation, in which different worker processes explore disjoint subsets of the candidate space. Another opportunity occurs in input generation and testing—*i.e.*, calling the same synthesized candidate on multiple inputs.

As scaling out involves constant overheads for process spawning and interprocess communication, scaling out makes sense only after constant costs are negligible relative to synthesis. This is achieved by having HARP scale out after a few AST levels have been explored.

6.3 Partial Regeneration

HARP may only partially regenerate L, if (1) a subset of library functions in L fail regeneration, *e.g.*, due to side-effects, or (2) if some developer tests—HARP's very last stage—fail. Partial regeneration can still be useful to developers in a variety of ways. For example, the regenerated library can operate side-by-side with a hardened version of the original library.

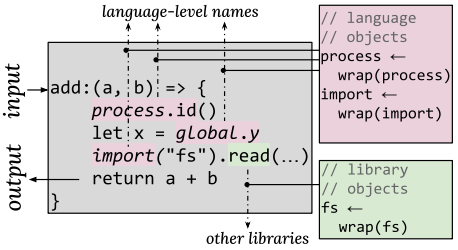
The latter fast-slow setup combines improved security properties with acceptable overall performance. The partially regenerated L' serves the majority of the calls, and it does so efficiently and securely. At times, however, L' receives input that falls outside its expected range of operation—but not outside that of L. These inputs result into a runtime exception, caught by a HARP controller component, which then forwards the input to L. As L now executes with additional hardening in place, it is significantly less efficient, but still computes the correct output securely. The exact hardening mechanism and thus its performance overhead can vary significantly [27, 33, 39, 65, 66], and depends directly on details related to the threat model—for example, native memory-unsafe binaries require additional care.



6.4 Quick Aborts

HARP implements a `--quick-abort` option that quits searching if HARP detects behaviors such as file-system and environment-variable access are not observable to clients that work only with values returned from the target library. Such behaviors signal that the original library may be falling outside HARP's model of computation, allowing HARP to quickly abort the ALR process.

To record library accesses to functionality implemented outside the library, HARP instruments all names that remain free at the top-level scope of the library—*i.e.*, ones that are not bound to values *in* the library. HARP starts from a few well-known root names—a static list of names provided by default by the language and runtime environment. For example, in server-side JavaScript these names include the `global` variable table, the `require` function for importing other libraries, and the `process` object for providing access to environment variables, process arguments, and other information in the broader environment.



Load-Time Transformations: Modern dynamic languages feature a module-import mechanism that loads code at runtime as a string. HARP applies lightweight load-time code transformations on the string representation of each module, as well as the context to

which it is about to be bound, to insert instrumentation wrappers into the module before it is loaded.

Harp's transformations first create a modified copy of a module's runtime context. The context is a name-value mapping for all free name variables available to the module by default. The modifications target the values in this mapping—traversing and wrapping each value with an interposition mechanism that records the access in a global access table. Harp then binds the modified context to the module, using a source-to-source transformation that redefines names in the context as library-local ones and assigns to them the values of the modified context.

Harp's transformations have a common structure that traverses objects recursively—a base transform `wrap`, which we review first (and whose effects are shown in Fig. 6a). The `wrap` transform takes an object O and returns a new object O', where every field f of O is wrapped with and replaced by a method f'. If called, f' adds a record to a global map noting that this particular field f has been accessed and then passes arguments to f.

Context Creation: To prepare a new context to be bound to a library being loaded, Harp first creates an auxiliary hash table (Fig. 6*b*), mapping names to newly transformed values: names correspond to implicit modules—globals, language built-ins, module-locals, *etc.*; transformed values are created by `wrap`ping individual values in the context to insert instrumentation hooks.

User-defined global variables are stored in a well-known location (*e.g.*, a map accessible through a global variable named `global`). However, traversing the global scope for built-in objects is generally not possible. To solve this problem, Harp collects such values by resolving well-known names hard-coded in a list. Using this list, Harp creates a list of pointers to unmodified values upon startup.

Care must be taken with module-local names such as the module's absolute filename, its `exported` values, and whether the module is invoked as the application's `main` module. These names refer to a different value for each module, and thus attempting to access the values directly from within Harp's transformation scope will fail subtly: the names will end up resolving to module-local values of Harp *itself*. Harp solves this issue by deferring these transformations to the context-binding phase (discussed next).

Context Binding: To bind the code whose context is being transformed with the freshly created context, Harp applies a source-to-source transformation that wraps the module with a function closure (Fig. 6c.). By enclosing and evaluating a closure, Harp leverages lexical scoping to inject a non-bypassable step in the variable name resolution mechanism.

The closure starts by redefining default-available non-local names as module-local ones, pointing to transformed values that exist in the newly-created context. It accepts as an argument the customized context and assigns its entries to their respective variable names in a preamble consisting of assignments that execute before the rest of the module. Module-local variables (a challenge outlined earlier) are assigned the return value of a call to `wrap`, which will be applied only when the module is evaluated and the module-local value becomes available. Harp evaluates the resulting closure, invokes it with the custom context as an argument, and applies further `wrap` transformations to its return value.

7 IMPLEMENTATION & EVALUATION

In summary, Harp's evaluation answers the following questions:

- **Q1: Can Harp eliminate real vulnerabilities?** Harp successfully eliminates vulnerabilities that enable 3 large-scale software supply-chain attacks (§7.2–7.4) by learning and regenerating the core functionality of the vulnerable library, eliminating any dependency to dangerous code. To the best of our knowledge, Harp is the first system that can eliminate these attacks.
- **Q2: How long does ALR take?** Applied to 17 JavaScript string-processing libraries (§7.5), Harp learns 14 libraries within a minute and all under an hour. It also aborts within 5 seconds on 11 other JavaScript libraries that fall outside the string-processing domain. Harp's domain-specific performance refinements (§6.2) improve the runtime performance of ALR by 179.27×.
- **Q3: What are the characteristics of regenerated libraries?** The regenerated libraries execute between 2% faster and 7% slower than the original JavaScript libraries. The regenerated libraries import nothing and use only basic JavaScript language primitives. The original libraries, in contrast, have access to the entire JavaScript ecosystem, including standard JavaScript and Node.js libraries, the file system, the network, environment variables, and process arguments.
- **Q4: Is ALR applicable outside JavaScript?** Harp successfully regenerates JavaScript versions of 5 native string-processing libraries (Appendix B). The regenerated libraries incur a maximum overhead of 1% and enjoy memory and type safety benefits not present in the original libraries.

7.1 Methodology

Workloads: To investigate Q1, we obtained 3 widely-publicized software supply-chain security incidents from the JavaScript ecosystem: (1) `event-stream` [41, 61], a popular library that was modified to steal bitcoins from specific Bitcoin wallets (§7.2), (2) `left-pad` [37, 69], a popular library replaced by a no-op after a package name dispute, breaking thousands of projects including Facebook and PayPal (§7.3); and (3) `string-compare` [10], where two different versions of the same string comparison library—one benign and one malicious—appear in the same dependency tree (§7.4).

To investigate Q2 and Q3, we obtained 14 additional JavaScript string processing libraries from `npm` with the help of an experienced JavaScript developer and a senior undergraduate student. The student used the `npm`'s search feature to search for libraries using a variety of string-processing terms such as "padding", "strip," and "change case." For each term, the student sorted the list of returned libraries by popularity [44] to inspect the first five pages of search results and select the library that provided the most complete corresponding functionality. We note that this process excludes duplicates—for example, the student found and discarded more than 10 `left-pad` libraries with similar or identical functionality. This phase produced 17 unique string-processing libraries that are used pervasively and can affect a large part of the ecosystem [72]: collectively, these libraries are directly imported by several applications and transitively imported via other dependencies by more than 100K applications. The phase also produced 11 libraries that were misclassified as string processing libraries. We applied Harp to all 28 libraries.

To investigate Q4, we obtained C/C++ libraries by searching GitHub using the same search terms as for the JavaScript libraries. Since many of these libraries did not have tests or client programs, we opted for C/C++ libraries with JavaScript bindings to check compatibility via tests and client programs from the JavaScript ecosystem. This search process produced five libraries.

Evaluation metrics: We evaluate *security* improvements qualitatively and quantitatively. For known attacks (Q1), we first used the original (compromised) library to reproduce the attack. We then inferred and regenerated the original library and replaced the original library with the regenerated version. We confirmed that the regenerated version eliminated the attack. For all libraries (Q2–5), we report the privilege reduction achieved after applying Harp. This quantitative security metric was developed recently [66] and corresponds to a ratio α/t, where α is the count of all APIs that are not invocable by the library any more, due to the defense applied, and t is the total count of APIs made available to a library by default by the combined built-in or third-party libraries.

We evaluate the *correctness* of regenerated libraries (Q3, Q5) using a combination of developer tests, client libraries or applications, and manual inspection. We ran the developer-provided test suites for the libraries and verified that the regenerated libraries provide correct results. We also imported the regenerated libraries into the top 10 client libraries or applications that directly import the original libraries and ran the test suites for these client libraries or applications. Finally, we manually inspected the regenerated code to confirm that it correctly implements the intended correct behavior of the original version.

For the *learning time* (Q2), we report wall-clock time after the call `npm-install` up to the point where Harp either (1) aborts, reporting intractability, (2) timeouts, failing to synthesize a library, or (3) succeeds, regenerating a library and its appropriate bindings. We set the timeout limit to 12 hours. We measured the *runtime performance* of regenerated-libraries (Q4, Q5) using a combination of developer tests and synthetic workloads operating in tight loops. We repeated all performance-related experiments 100 times and report averages.

Implementation Details: Harp currently works with black-box libraries available in JavaScript, Python (not shown here; reported in the extended version [blind]), and binary object files developed, for example, in C/C++ and wrapped as native add-ons. We expect native add-ons to be wrapped by some form of language-level interface such as Node's NaN or N-API and Python's `ctypes` or CFFI. Harp's ALR components, including the synthesis and DSL, are written in JavaScript. The base set of DSL terms as well as the resulting programs are compiled to their respective language using a small Python compiler: the compiler currently can emit JavaScript and Python programs, which are then executed using the interpreter of the respective language. The regenerated programs link against a small utility library that provides runtime support, ported once for each target language supported by Harp.

Harp currently has a few limitations. First, it does not support libraries whose functions mutate built-in, prototype, or other objects—such as `String.prototype` in JavaScript. Additionally, Harp's input generation algorithm does not generate non-ASCII strings or ones with special—possibly hierarchical—structure such as JSON, HTML, and CSS; generating the latter without any additional domain information would be impractical.

Software and Hardware Setup: All experiments were conducted on a server with 512GB of memory and 64 physical × 2.1GHz Intel Xeon E5-2683 cores, running Debian 4.9.144-3.1. The JavaScript setup uses Node.js v12.19, bundled with V8 v7.8.279.23, LibUV v1.39.0, and `npm` version v6.14.8; the Python setup uses CPython 3.7.5. To perform timeline-accurate supply-chain attacks, we set up a private registry using `verdaccio` [64] available only to the server running the experiments.

7.2 Use Case: Event-Stream

The `event-stream` incident [41, 61], discussed extensively earlier (§2), introduced a malicious dependency harvesting Bitcoin account credentials through a popular stream-processing library. This dependency, `flatmap-stream`, targeted a very specific production environment of a cryptocurrency application; other environments were not affected.

Security: We reconstruct the malicious library and payloads from a variety of sources [21, 45, 52]. The library applies several checks to verify it runs on production, as part of a specific application, and as part of a specific build. If all these conditions hold, it then writes to the file-system. Harp's active learning phase does not infer any file-system accesses—because there are no such accesses during the learning phase and because the Harp DSL does not include file system operations. As a result, Harp regenerates an exploit-free version of the library, confirmed by manual inspection. It makes no use of built-in APIs, achieving a privilege-reduction of 332×.

Performance & correctness: Harp takes on average 1.4 seconds to complete `flatmap-stream`'s active learning and regeneration. We manually inspected the regenerated code and found it implements the full functionality of the original library. The original library does not come with any test cases and the version of `event-stream` that uses the malicious `flatmap-stream` version has been removed permanently from npm. We therefore manually modified `event-stream` commit `e316336`, introducing `flatmap-stream` to import the regenerated `flatmap-stream`, and apply `event-stream`'s tests. All 14 (100%) of `event-stream`'s tests pass successfully: 13/14 tests are not affected by the `flatmap-stream` addition, and 1/14 that tests `flatmap-stream` passes successfully. Applying the regenerated `flatmap-stream` to an array of 1000 elements over 10K runs takes 48.99 seconds—an overhead of about 107μs per run over the performance of the original library.

7.3 Use Case: Left-Pad

The `left-pad` incident [37, 69] was caused by unpublishing a popular JavaScript library, effectively replacing it permanently with a No-Op. While `left-pad` itself was an 11-line moderately-popular string-padding function, it was used by many popular projects such as React and Babel. The unpublishing corrupted production environments, denying them the ability to revert to an older version of the library. As a result, the incident affected more than one third of the Node.js ecosystem, and led to significant changes in the un-publishing policies of public library registries.

Security: We apply Harp to an identical library built by npm as a response to the incident, replacing the original left-pad library copied to our local registry (§7.1). Harp regenerates all of left-pad's functionality, fully eliminating the dependency. As a result, left-pad's tests still succeed after we unpublish left-pad from our local registry because they no longer depend on the original left-pad module. The regenerated left-pad makes no use of built-in APIs, resulting in a privilege-reduction score of 332×.

Performance & correctness: Harp completes left-pad's active learning and regeneration in an average of 3.6 seconds. We manually inspect the regenerated code and confirm it implements left-pad's full functionality. We apply the full test suite (35 tests) from left-pad's repository, all of which (100%) pass successfully. One test is particularly interesting as it supplies ill-defined input to trigger left-pad's default behavior, by providing a padding length of false, it invokes one of left-pad's padding behaviors that Harp learned through other false-y values. The runtime performance of the regenerated left-pad on 10K runs is 45.66 seconds—an overhead of about 20μs per run over the performance of the original library.

7.4 Use Case: String-Compare

The string-compare attack involves two versions of a single library in the same codebase [10]. An earlier version of this library, used as part of a sort function, is benign. A later version, used in the authentication module, is malicious: if provided the (authentication) string gbabWhaRQ, it access the file system of the server running the program.

Security: We apply Harp to both versions of string-compare library. The regenerated version is identical in both cases. It does not contain any side-effects—nor the check that launches the attack in the second case. Harp eliminates the string-compare dependency from both sort and auth, replacing it with vulnerability-free code. The regenerated string-compare makes no use of built-in APIs, resulting in a privilege-reduction score of 332×.

Performance & correctness: Harp completes string-compare's active learning and regeneration in an average of 0.7 seconds. We manually inspect the regenerated code and confirm it implements string-compare's full functionality. As string-compare comes with no test cases, we apply the test cases of the sort function over a shuffled version of Ubuntu's wamerican dictionary words file (102K elements); all 3 (100%) test cases pass. The runtime performance of 10K sort iterations using the regenerated string-compare takes 46.01 seconds—an overhead of about 41μs per run over the performance of the original library.

7.5 Applying Harp to More Libraries

In this section, we apply Harp to 25 JavaScript libraries—17 string-processing libraries and 11 other libraries—and 5 C/C++ libraries collected from GitHub.

Table 1 shows results for 17 JavaScript string-processing libraries. The statistics columns D_1–D_4 count the number of weekly downloads, direct dependents, total dependents, and direct dependencies of these libraries as reported by the npm tool: collectively, these libraries can affect a significant fraction of the ecosystem—they total 102M downloads per week, are directly depended upon by a total of 4.3K libraries, and are indirectly depended upon by more than 15K libraries and applications. The Learning columns t_1 and t_2 show the time it took Harp to apply active learning and regeneration; t_1 is full Harp, whereas t_2 does not include Harp's performance refinements (§6). The Regeneration columns Performance and Correctness show the characteristics of the regenerated library with respect to the original: Performance is measured using 10K iterations of several tests; and Correctness is measured by running all the tests of the original library and 10 client-libraries against the regenerated library, followed by manual inspection.

Learning: Harp's active learning and regeneration takes between 0.7 seconds and 50.9 minutes (avg.: 204.83 seconds) to complete, with 14 out of 17 libraries regenerated within a minute and 16 out of 17 libraries regenerated within 145 seconds (2.4 minutes). The regenerated camel-case library stands out in terms of size, containing 8 computational statements, two of which are split operators, taking 3059 seconds (50.9 minutes) to regenerate.

Harp's performance refinements (§6) offer significant improvements. Without any refinements, Harp takes at least 179.27× longer. This value is a conservative estimate because Harp reaches a timeout limit of 12 hours for 7 out of 17 libraries. This speedup includes a 1.1× slowdown for 5 small libraries that are penalized by Harp's refinements. As the regeneration of these libraries remains within a few seconds, their slowdown is considered acceptable—especially given the overall speedup of long-running regenerations.

In terms of code coverage, the input generation algorithm exercises 100% of library code for 11 out of 17 libraries. For the remaining six libraries, the majority of the functionality not exercised is related to exception handling. In the case of decamelize (80%), Harp does not exercise four lines handling erroneous input (non-string arguments), and 11 lines related to a flag preserving consecutive upper case. In the case of flatmap-stream (62.2%), Harp does not exercise a subset of the stream-specific functionality—stream pause, resume, destroy and end handlers—that are part of superclass functionality. Harp also does not exercise exception handlers in the write method, which are meant to handle errors propagating up from the stream consumer. In the case of repeat-string (95.45%), Harp misses an exception-raising statement meant to cover cases where the first argument is not a string. The trim library (33.3%) first checks if the input string's prototype includes a trim method and if so it invokes it; otherwise, implements a left and right trim by invoking other methods—but Harp's primary inputs always support trim as part of the string prototype. In the case of upper-case (44%), Harp misses all the locale-specific code (confirmed by the tests). In the case of zero-fill (80%), Harp misses a branch for when the second argument (the "filler" string) is not provided—in which case the library returns a partially applied function.

Performance: To understand the runtime performance of regenerated libraries, we apply them on the tests of the original libraries in tight loops of 10K iterations. Their performance is between -1.6% (speedup) and 6.4% (slowdown), with an average of 2.3%. Profiling shows that the overhead comes from Harp's complex pattern matching primitives which compile down to the language's regular-expression language (REL). REL is in fact not regular, as it supports back-references and other non-regular constructs, and thus does not perform as efficiently as the simpler string-matching constructs found in the original libraries.

Tab. 1: Applying Harp to JavaScript libraries. Columns D_{1-4} show weekly downloads, direct dependents, total dependents, and direct dependencies; columns t_1 and t_2 show the time it took to complete ALR, with and without refinements; column *Coverage* shows the percentage of the source code covered by the input generation algorithm; other columns show the characteristics of the regenerated libraries compared to the original libraries.

	Popularity				Learning			Regeneration			
Library	D_1	D_2	D_3	D_4	t_1 (s)	t_2 (s)	Cov/ge (%)	Perf/nce (s)	(%)	Correctness (%)	
camel-case	13,007,997	603	1484	2	3059	>12h	100	10.15	(5.6%)	9/9	(100%)
constant-case	3,104,230	88	1651	3	22	>12h	100	9.86	(3.8%)	9/9	(100%)
decamelize	22,883,567	1042	1132	0	4	14340	80	9.61	(2.9%)	40/40	(100%)
flatmap-stream	70	0	0	0	1.4	>12h	71.21	48.99	(2.2%)	14/14	(100%)
left-pad	3,077,112	509	518	0	3.6	1.3	100	45.66	(0.4%)	35/35	(100%)
no-case	13,051,868	114	2463	2	28	>12h	100	9.89	(5.3%)	31/31	(100%)
pascal-case	8,995,694	540	2979	2	145	>12h	100	10.15	(6.4%)	8/8	(100%)
repeat-string	17,921,038	580	832	0	19	8	95.45	9.18	(-1.6%)	28/30	(93.3%)
sentence-case	2,809,582	58	1636	3	129	>12h	100	10.02	(4.9%)	7/7	(100%)
snake-case	3,045,948	293	1386	2	36	>12h	100	9.95	(3.4%)	8/8	(100%)
string-compare	46	0	0	1	0.7	1.2	100	46.01	(0.9%)	3/3	(100%)
trim-left	789	6	8	0	3	3.5	100	9.36	(0.3%)	4/4	(100%)
trim-right	3,874,815	200	249	0	2	3.2	100	9.43	(2.9%)	4/4	(100%)
trim	3,684,237	206	472	0	21	111	33.33	9.45	(1.2%)	4/4	(100%)
upper-case	6,888,443	115	331	1	3	1.9	44.44	9.38	(0.0%)	4/6	(66.7%)
write-pad	15	0	0	1	3	1.4	100	9.26	(0.9%)	1/10	(100%)
zero-fill	35,143	43	175	0	3	1.6	80	9.27	(-0.9%)	11/16	(68.8%)
Min	15	0	0	0	0.7	1.2	33.33		(-1.6%)		(66.7%)
Max	22,883,567	1042	2979	3	3K	>12h	100		(6.4%)		(100.0%)
Avg	6,022,388	258.6	900.9	1	204.83	>10.2h	86.04		(2.3%)		(95.8%)

Correctness: Out of 17 libraries, 14 are fully regenerated, passing 100% of developer-provided and client-application tests. Three libraries are partially regenerated, passing between 4/6 (66.7%) and 28/30 (99.3%) of tests. Two of `repeat-string`'s tests are designed to generate exceptions—not expressible in Harp's DSL. Two of `upper-case`'s tests are locale-dependent, with string locales that fall outside Harp's input generation algorithm. Finally, 5 of `zero-fill`'s tests expect a partially evaluated function, which is currently not expressible in Harp's DSL.

8 RELATED WORK

Active Learning: Active learning is a classical topic in machine learning [54]. Recent research has introduced the concept of using a domain-specific language to define and infer classes of target computations [9, 50, 56, 57]. This approach promotes the clean identification of target computations across a range of domains and the development of efficient inference algorithms for these computations. The approach has been used to infer and regenerate computations with additional safety checks, increased functionality, enhanced user interfaces, the ability to operate in new execution contexts, expressed in different programming languages, and to synthesize combine operations that enable the exploitation of divide and conquer parallelism in stream computations [9, 50, 55–57, 67]. Example application domains include data parallelism in Unix shell scripts [55], programs that access databases [56, 57], work with key/value stores [50], or interact with external or internal components [9, 67]. Functionality developed in this context includes an efficient top-down inference algorithm [56, 57], and algorithms that use input shapes to generate inputs that productively disambiguate potential candidate inferred computations (Harp and [55]). Harp is the first active learning and regeneration system to target string computations and the first to successfully eliminate software supply-chain vulnerabilities in widely used libraries.

Input-Output Synthesis: Program synthesis and programming by example automatically generate programs that satisfy a given set of input-output examples [2, 15, 17, 18, 22, 47, 48, 59, 68]. Harp differs in that it works with an existing component as opposed to a fixed set of input-output examples and interacts with the component to build a model of its behavior. The goal is to eliminate dependencies and vulnerabilities by replacing the original version with the regenerated version, without requiring developers to provide input-output examples manually.

Component-based synthesis [14, 16, 32, 58] aims to generate a program consisting of library calls to a provided API. It synthesizes code for making library calls by executing the candidate program on a set of test cases. Harp, in contrast, infers and regenerates complete string computations. Instead of working with a provided set of test cases, Harp uses active learning to automatically and adaptively generate custom inputs that enable Harp to infer each string computation.

Component Protection: Runtime component protection techniques provide monitoring, instrumentation, and policy enforcement, typically through sandboxing, wrapping, or transformation [1, 19, 24, 31, 34–36, 38, 51, 53, 63]. Harp differs in that it replaces the library with a regenerated version instead of executing the library in a sandbox or wrapping the library to dynamically enforce a security policy. The regenerated library therefore executes with no runtime instrumentation overhead and requires no sandboxing. We note that, to avoid exploitation during inference, Harp uses a combination of sandboxing and wrapping during inference. Unlike sandboxing or wrapping, Harp also protects against library deletion attacks.

Software Debloating: Software debloating [3, 20, 25, 26] lowers the potential for vulnerabilities by eliminating unused code in a program. Functionality excision [49] removes code that implements counterproductive, irrelevant, or undesirable functionality. Like debloating and functionality excision, Harp can eliminate code in the inferred library. Unlike debloating, which prunes computation in the original library but leaves unpruned code intact, Harp replaces the original library with a regenerated version that is guaranteed to conform to a safe model of computation. Harp can also discard potentially malicious code in the regeneration, including code that executes during inference but does not affect the client-visible behavior of the inferred library.

Vulnerability Detection: Prior work on static [8, 12, 13, 23, 62] and dynamic [46, 70] analysis can detect malicious code at development or production time. Harp does not attempt to detect a vulnerability—rather, it assumes libraries as a potential liability with stealthy Turing-complete vulnerabilities, and rewrites them into functionally equivalent, side-effect-free versions.

9 DISCUSSION & LIMITATIONS

Synthesis Limitations: There are desirable guarantees that Algorithm 1 does not satisfy. First, if the behavior of the original library L does not correspond to any program in the DSL, there is no guarantee that the algorithm will determine this fact in any finite time—it is possible that the difference in behavior will be exposed only by an input that the algorithm has yet to consider. Second, if the behavior of the original library L does correspond to some program in the DSL, there is no guarantee that the algorithm will find that DSL program in any finite time—it is possible that the program is larger than programs that the algorithm has considered.

So given a set of DSL programs P at some point in execution of Algorithm 1, what must be true of the relationship between the DSL programs $f \in P$ and the original library L? First, P and all f exhibit identical behavior on all considered inputs I. Second, if there is some program f of size n or less that has the same behavior as L on all inputs, then $f \in P$. If additionally $P = \{f\}$, then the algorithm will return a new library L' that has identical behavior as the original library L. There are two key preconditions here which Algorithm 1 does not check: (1) there is some DSL program f which has identical behavior on all inputs as the original library L, and (2) this DSL program is of size n or less for some known n. These preconditions may come, for example, from the general domain knowledge of the programmer.

Attacks on outputs: As outlined earlier (§3), the primary targets of active library learning and regeneration are (1) side-effectful attacks—*e.g.*, ones targeting the file system, global variables, the module system, process arguments, or environment variables, (2) attacks via low-level, memory-unsafe, and type-unsafe code such as ones typical in C and C++ code. Could ALR additionally protect against attacks targeting the output of a library function?

For attacks that target function outputs, there are two broad possibilities. If the malicious behavior is hidden and therefore not exposed during testing/normal use, Harp will not learn the malicious behavior and thus the regenerated code will not contain the corresponding vulnerability. If, however, the malicious behavior is exposed during testing/normal use, Harp would either (1) determine that the observed behavior is outside the scope of the DSL and reject the library, or (2) learn and regenerate the behavior. In the latter case, Harp is relying on the exposure of the malicious behavior during testing/normal use to detect and eliminate the behavior—*i.e.*, we would expect the behavior to be detected by the developer during development and before deployment. We anticipate that, at least for string processing programs, almost all such malicious behaviors will be outside the scope of the Harp DSL.

Generalizing ALR: Active learning and regeneration is a black-box program inference approach that fixes (1) a specific computational domain (SCD) such as string processing, tensor operators, database interfaces, (2) a corresponding language (DSL) for modeling computations in that domain, and (3) an input generation algorithm (IGA) for interacting with the black-box computation. These three elements are interlinked and are designed to complement each other. For example, the DSL is designed to enable differential testing, using the IGA to guide efficient inference—by inferring the existence or absence of certain DSL terms in the regenerated programs while minimizing ambiguity.

An active learning and regeneration *system* such as Harp is an instantiation of these three elements (SCD, DSL, IGA) for a particular domain. We do not expect a single system to be expanded to capture all or even a large range of computation of interest. Such an expansion can quickly result in general computations and thus quickly hit known intractability limits.

Applying ALR to further domains: Instead, we anticipate multiple active learning and regeneration systems, each targeting a certain class of libraries. Harp exemplifies this approach for string computations—a central, widely used class of computations. Other classes of computation and associated DSLs include: arithmetic-operation libraries, linear algebra and tensor operations, key-value operations, spreadsheet-style computations, components that access SQL databases, blockchain smart contracts, and stream-based parallelizing combiners.

The technique has been applied successfully in some of these domains—*e.g.*, programs that access stateful key-value stores [50], applications that access relational databases [56, 57], binary data parsing and transformation [9], and synthesis of parallel Unix shell commands [55]. Harp is the first active learning and regeneration system to target security vulnerabilities in software supply chains.

10 CONCLUSION

Supply-chain attacks are becoming a critical security concern. This paper presented a new approach, active library learning and regeneration (ALR), to infer and regenerate the client-observable functionality of a black-box, third-party software dependency. The regenerated dependency leverages domain-specific modeling, in which the target class of attacks cannot be expressed. We demonstrate ALR in Harp, a prototype system for inferring and regenerating components that implement string computations.

ACKNOWLEDGMENTS

We thank Dimitris Karnikis, Grigoris Ntousakis, Kostis Sagonas, the participants of 18th Athens PL Seminar, and our shepherd Sooel Son. This research was funded in part by DARPA contracts HR00112020013 and HR001120C0191.

REFERENCES

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete Client-side Sandboxing of Third-party JavaScript Without Browser Modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2420950.2420952

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.

[3] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1697–1714.

[4] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. 2014. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 64–83. https://tweetnacl.cr.yp.to/

[5] Oscar Bolmsten. 2017. Malicious Package: crossenv and other 36 malicious packages. https://snyk.io/vuln/npm:crossenv:20170802 Accessed: 2019-03-19.

[6] Benjamin Byholm, Rod Vagg, and NAN contributors. 2018. Native Abstractions for Node. https://www.npmjs.com/package/nan Accessed: 2020-06-11.

[7] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 516–519.

[8] Stefano Calzavara, Michele Bugliesi, Silvia Crafa, and Enrico Steffinlongo. 2015. Fine-Grained Detection of Privilege Escalation Attacks on Browser Extensions. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 510–534. https://doi.org/10.1007/978-3-662-46669-8_21

[9] José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard. 2019. Active Learning for Software Engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 62–78. https://doi.org/10.1145/3359591.3359732

[10] David Bryant Copeland. 2019. The Frightening State of Security Around NPM Package Management. https://bit.ly/3pID2h1 Accessed: 2020-12-10.

[11] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. NDSS.

[12] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 257–269. https://doi.org/10.1145/3359789.3359813

[13] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Springer International Publishing, Cham, 303–325.

[14] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 599–612.

[15] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 229–239.

[16] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*. 653–663.

[17] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM Sigplan Notices*, Vol. 46. ACM, 317–330.

[18] Shivam Handa and Martin C. Rinard. 2020. Inductive program synthesis over noisy data. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Nov 2020). https://doi.org/10.1145/3368089.3409732

[19] Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld. 2017. A principled approach to tracking information flow in the presence of libraries. In *International Conference on Principles of Security and Trust*. Springer, 49–70.

[20] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394.

[21] hugeglass. 2018. GitHub Repository for flatmap-stream. https://git.io/Jtcdi Accessed: 2020-12-18.

[22] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 215–224.

[23] N. Jovanovic, C. Kruegel, and E. Kirda. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S P'06)*. 6 pp.–263. https://doi.org/10.1109/SP.2006.29

[24] Yoonseok Ko, Tamara Rezk, and Manuel Serrano. [n. d.]. SecureJS Compiler: Portable Memory Isolation in JavaScript. In *SAC 2021-The 36th ACM/SIGAPP Symposium On Applied Computing*.

[25] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*.

[26] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security*. 1–6.

[27] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS'17)*. ACM, New York, NY, USA, 51–57. https://doi.org/10.1145/3144555.3144562

[28] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. (2017).

[29] SS Jeremy Long. 2015. OWASP Dependency Check. (2015).

[30] Michael Maass. 2016. *A Theory and Tools for Applying Sandboxes Effectively*. Ph.D. Dissertation. Carnegie Mellon University.

[31] Jonas Magazinius, Daniel Hedin, and Andrei Sabelfeld. 2014. Architectures for inlining security monitors in web applications. In *International Symposium on Engineering Secure Software and Systems*. Springer, 141–160.

[32] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. *ACM Sigplan Notices* 40, 6 (2005), 48–61.

[33] Marcela S Melara, David H Liu, and Michael J Freedman. 2019. Pyronia: Redesigning Least Privilege and Isolation for the Age of IoT. *arXiv preprint arXiv:1903.01950* (2019).

[34] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 481–496.

[35] James Mickens. 2014. Pivot: Fast, synchronous mashup isolation using generator chains. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 261–275.

[36] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2009. Caja: Safe active content in sanitized JavaScript, 2008. *Google white paper* (2009).

[37] Paul Miller. 2016. How an irate developer briefly broke JavaScript. https://bit.ly/36CkBDI Accessed: 2020-12-10.

[38] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. 2019. ScriptProtect: mitigating unsafe third-party javascript practices. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 391–402.

[39] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 699–716.

[40] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 736–747.

[41] npm, Inc. 2018. Details about the event-stream incident. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident Accessed: 2018-12-18.

[42] npm, Inc. 2019. Malicious Package: stream-combine. https://www.npmjs.com/advisories/774 Accessed: 2019-01-25.

[43] npm, Inc. 2019. Malicious Package: stream-combine. https://www.npmjs.com/advisories/765 Accessed: 2019-01-25.

[44] npm, Inc. 2020. Node Package Manager. https://www.npmjs.com/search?q=string&ranking=popularity

[45] Jarrod Overson. 2018. BadJS—Malicious JavaScript found in the wild: Event-Stream. https://badjs.org/posts/event-stream/ Accessed: 2020-12-18.

[46] Giancarlo Pellegrino and Davide Balzarotti. 2014. Toward Black-Box Detection of Logic Flaws in Web Applications.

[47] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

[48] Mohammad Raza and Sumit Gulwani. 2018. Disjunctive Program Synthesis: A Robust Approach to Programming by Example. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[49] Martin Rinard. 2011. Manipulating program functionality to eliminate security vulnerabilities. In *Moving target defense*. Springer, 109–115.

[50] Martin C. Rinard, Jiasi Shen, and Varun Mangalick. 2018. Active Learning for Inference and Regeneration of Computer Programs That Store and Retrieve Data. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas,*

New Paradigms, and Reflections on Programming and Software (Onward! 2018). ACM, New York, NY, USA, 12–28. https://doi.org/10.1145/3276954.3276959

[51] José Fragoso Santos and Tamara Rezk. 2014. An information flow monitor-inlining compiler for securing a core of javascript. In *IFIP International Information Security Conference*. Springer, 278–292.

[52] Thomas Hunter II (Intrinsic Security). 2018. Compromised npm Package: event-stream. https://medium.com/intrinsic/compromised-npm-package-event-stream-d47d08605502 Accessed: 2019-03-19.

[53] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. 2003. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/945445.945448

[54] Burr Settles. 2009. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison.

[55] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. 2021. Automatic Synthesis of Parallel Unix Commands and Pipelines with KumQuat. *CoRR* abs/2012.15443 (2021). arXiv:2012.15443 https://arxiv.org/abs/2012.15443

[56] Jiasi Shen and Martin C. Rinard. 2019. Using Active Learning to Synthesize Models of Applications That Access Databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 269–285. https://doi.org/10.1145/3314221.3314591

[57] Jiasi Shen and Martin C. Rinard. 2021. Active Learning for Inference and Regeneration of Applications That Access Databases. *ACM Trans. Program. Lang. Syst.* 42, 4, Article 18 (Jan. 2021), 119 pages. https://doi.org/10.1145/3430952

[58] Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[59] Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment* 9, 10 (2016), 816–827.

[60] Snyk. 2016. Find, fix and monitor for known vulnerabilities in Node.js and Ruby packages. https://snyk.io/

[61] Ayrton Sparling et al. 2018. Event-Stream, GitHub Issue 116: I don't know what to say. https://github.com/dominictarr/event-stream/issues/116 Accessed: 2018-12-18.

[62] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Networked and Distributed Systems Security (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23071

[63] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An empirical study of information flows in real-world javascript. In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*. 45–59.

[64] Trent Earl, John Wilkinson, and the Verdaccio contributors. 2018. Verdaccio—npm Proxy Private Registry. https://verdaccio.org/ Accessed: 2020-11-10.

[65] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Networked and Distributed Systems Security (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23131

[66] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 18. https://doi.org/10.1145/3460120.3484535

[67] Jerry Wu. 2018. *Using dynamic analysis to infer Python programs and convert them into database programs*. Master's thesis. Massachusetts Institute of Technology.

[68] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated migration of hierarchical data to relational tables using programming-by-example. *Proceedings of the VLDB Endowment* 11, 5 (2018), 580–593.

[69] Serdar Yegulalp. 2016. How one yanked JavaScript package wreaked havoc. https://bit.ly/3ofwkz2 Accessed: 2020-12-10.

[70] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. Association for Computing Machinery, New York, NY, USA, 116–127. https://doi.org/10.1145/1315245.1315261

[71] Nicholas C. Zakas and ESLint contributors. 2013. ESLint—Pluggable JavaScript linter. https://eslint.org/ Accessed: 2018-07-12.

[72] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Smallworld with High Risks: A Study of Security Threats in the Npm Ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, USA, 995–1010.

A PROOF SKETCHES

Definition A.1. **(IO-Correctness)** Given a function f, the synthesized function f' is said to be IO-correct, if and only if, f' is expressible in the Harp DSL (with constants extracted from f) and for all input i consistent with the input type of f, $f(i) = f'(i)$.

Definition A.2. **(Consistentency w.r.t. function f, input set I, and maximum program size n)** A synthesized function f' is said to be consistent w.r.t. a function f, input set I of size m containing inputs consistent with the input type of f, and a maximum program size n, if f' is expressible in the Harp DSL (with constants extracted from f) and is of size less than equal to n, and for all inputs $i \in I$:

$$f(i) = f'(i)$$

Note that, given a function f, the IO-correct function f' is consistent w.r.t. function f, for any input set I (consistent with the input type of f), and any maximum program size n greater than the size of f'.

Theorem A.3. **(Initial State)** *For any function f, all functions f' in P_n are consistent w.r.t function f, input set $I = \emptyset$, and maximum program size size n. Also, if there exists a function f' of size less than equal to n, which is IO-correct with respect to f, then $f' \in P_n$.*

Proof. The Harp algorithm extracts all constants from function f and instantiates all *sketches* is the Harp DSL of size n. The Type T (extracted using typeConstraints) is a *sound* approximation of the actual output type of f. A function $f' \in P_n$ if and only if f' of size less than equal to n, is expressible in the Harp DSL (with constants extracted from f), and T is a sound approximation of f''s output type. Therefore, given $I = \emptyset$, all functions in $f' \in P_n$ are consistent with f (input set $I = \emptyset$ and max size n).

Also, if there exists a IO-correct function f' of size less than equal to n, then f' is consistent with respect to f ($I = \emptyset$ and max size n) and T is a *sound* approximation of the output type of f'. Therefore, if there exists a IO-correct function f' of size less than equal to n, then $f' \in P_n$. □

Theorem A.4. **(Consistency)** *Given a function $f \in L$, let I be the set of inputs returned by the function* generateInputs*, P_n be the set of programs returned by* allPrograms*, and P be the set of pruned program* pruneSpace*. If $P \neq \emptyset$ and f' is equal to* getOpt(P)*,then f' is consistent w.r.t. function f, input set I, and maximum program size n. Also, if the IO-correct function $f' \in P_n$, then $f' \in P$.*

Proof. pruneSpace only prunes a function $f' \in P_n$ if and only if $\exists i \in I$, such that $f'(i) \neq f(i)$. Therefore, all $f' \in P$ are consistent with respect to f (input set I and max size n). The getOpt returns a function $f' \in P$, therefore if the algorithm synthesis a function f' for function f, then f' is consistent with respect to f (input set I and max size n).

pruneSpace will never prune out the IO-correct function f' as for all inputs $f(i) = f'(i)$. Therefore, if $f' \in P_n$, then $f' \in P$. □

Theorem A.5. **(Convergence)** *Given a function f and a maximum function size n, let F_n be the set of functions in the Harp DSL of size less than equal to n, such that, a IO-correct function $f' \in P_n$. As we add more inputs to the set of inputs generated by function* generateInputs*, Harp will synthesize a function f'', such that, f'' and f have the same output on an increasing set of inputs.*

Proof. P_n is equal to the set returned by allPrograms(n, T). From Theorem A.3, $f' \in P_n$. Let I be the set of input set constructed by generateInputs. Let P_I be the set of programs returned by the function pruneSpace. Note that if $f' \in P$, then for all I, $f' \in P_I$ (Theorem A.4).

Note that, if $I_0 \subseteq I_1$, then $P_{I_1} \subseteq P_{I_0}$ (as for any function $f'' \in P_{I_1}$, then f'' has the same output as f on inputs in I_0).

A larger set of inputs allows Harp to prune out functions which do not have the same output as f on this larger set of inputs. Therefore, by adding more inputs, Harp will synthesize a function f'', such that, f'' and f have the same output on an increasing set of inputs. □

B ADDITIONAL EVALUATION RESULTS

Non-string-processing Libraries: We also apply Harp on 11 libraries that were misclassified as processing strings, to evaluate Harp's `--quick-abort` mechanism. On these libraries, Harp aborts ALR within 5 seconds with a warning that they contain side-effectful computations that cannot be learned. Eight of these libraries import built-in modules that are not supported by Harp such as `debug`, `http`, or `fs`—for example, `minimatch` depends on `fs` and is thus not inferable. One of these libraries, `chalk`, depends indirectly on `os` and `tty` for checking the environment for color support and thus it not inferable. Finally, `ignore` and `attn` provide their functionality by extending the runtime context with an auxiliary value.

L	ALR	P(L')	C(L')
string-upper	2.9s	1.3%	66.7%
right-trim	2.7s	1.8%	100%
left-trim	2.6s	0.7%	100%
lr-trim	46.7s	0.4%	100%
repeat-text	17.1s	0.7%	100%

Fig. 7: C/C++ ALR. Harp applied to C/C++ libraries.

C/C++ Libraries: Fig. 7 summarizes results of applying Harp to 5 C/C++ libraries, including the time to complete learning (column ALR), the regenerated-library performance (column P(L') with positive values for slowdown and negative for speedup), and its correctness with respect to the original one (column C(L'), counting percentages of test cases). These libraries export a single function and are wrapped with Node's NAN module [6]. (NAN is an abstraction layer meant to simplify the development and maintenance of native add-ons over a constantly changing V8 API.)

Harp's ALR ranges between 2.6–17.1s (avg.: 14.4s), driven by the size of the regenerated library. Naturally, the performance of the regenerated JavaScript libraries is lower that that of the original compiled libraries, and ranges between 0.4–1.8% (avg.: 1.0%) of the original library's runtime performance (Col. P(L')). Harp regenerated full library behavior, except `string-upper`'s locale-dependent functionality.