Effective Error-Specification Inference via Domain-Knowledge Expansion

Daniel DeFreez
University of California, Davis, USA
dcdefreez@ucdavis.edu

Haaken Martinson Baldwin
University of California, Davis, USA
hmbaldwi@ucdavis.edu

Cindy Rubio-González
University of California, Davis, USA
crubio@ucdavis.edu

Aditya V. Thakur
University of California, Davis, USA
avthakur@ucdavis.edu

ABSTRACT

Error-handling code responds to the occurrence of runtime errors. Failure to correctly handle errors can lead to security vulnerabilities and data loss. This paper deals with error handling in software written in C that uses the return-code idiom: the presence and type of error is encoded in the return value of a function. This paper describes EESI, a static analysis that infers the set of values that a function can return on error. Such a function error-specification can then be used to identify bugs related to incorrect error handling. The key insight of EESI is to bootstrap the analysis with domain knowledge related to error handling provided by a developer. EESI uses a combination of intraprocedural, flow-sensitive analysis and interprocedural, context-insensitive analysis to ensure precision and scalability. We built a tool ECC to demonstrate how the function error-specifications inferred by EESI can be used to automatically find bugs related to incorrect error handling. ECC detected 246 bugs across 9 programs, of which 110 have been confirmed. ECC detected 220 previously unknown bugs, of which 99 are confirmed. Two patches have already been merged into OpenSSL.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; **Error handling and recovery**; **Software defect analysis**.

KEYWORDS

error handling, static analysis, bug finding

ACM Reference Format:
Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V. Thakur. 2019. Effective Error-Specification Inference via Domain-Knowledge Expansion. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338960

1 INTRODUCTION

Error-handling code responds to the occurrence of runtime errors in software. For example, a function attempting to allocate memory needs code to handle the case when there is no memory available, and a device driver requires code to handle the situation when the hardware device does not respond. Incorrect handling of such errors can lead to serious problems such as security vulnerabilities and data loss. Ignoring the error returned by a memory allocator would lead to the code accessing invalid memory, and ignoring the error returned by the hardware device might lead to data corruption. Thus, correctly handling errors in code is paramount.

This paper deals with error handling in software written in the C programming language. In the absence of exception-handling mechanisms, such C programs use the *return-code idiom*: the presence and type of error is encoded in the value returned by a function. Failure to correctly check for such error values at function call sites can lead to error-handling bugs (Section 4).

This paper presents *Effective Error-Specification Inference* (EESI; pronounced *ee-see*), a static analysis that infers function error-specifications for programs using the return-code idiom. A *function error-specification* is the set of values that the function can return on error. For example, the function `acpi_pci_link_allocate_irq` in the Linux kernel returns a negative integer on error, `acpi_ec_alloc` returns 0 on error, and `acpi_allocate_root_table` returns a nonnegative integer on error.

To understand the challenges in inferring error specifications, consider the following (intentionally abstract) code:

```
int f1() {
  if (f2() < 0) {
    f3();
    return 0;
  }
  return 1;
}
```

The function `f1` has only two possible return values `0` and `1`. Looking at the body of the function `f1`, we cannot infer whether `f1` returns `0` or `1` on error, or whether it is infallible (does not return any error value). There is nothing inherent in the code that implies that a particular value represents an error. Contrast this situation with that in languages supporting exception handling: exceptions are caught and handled in well-labeled `catch` blocks.

To tackle this challenge, EESI bootstraps the analysis by utilizing developer-provided domain knowledge. Such domain knowledge can include a list of functions that only occur along error paths (*error-only functions*), a list of *error codes*, or the error specification for a few functions (Section 3.2). For example, if the domain knowledge includes the fact that calls to function `f3` only occur along error paths, then EESI will infer that `f1` returns `0` on error; if the domain knowledge provided includes the fact that `f2` returns a nonnegative value on error, then EESI will infer that `f1` returns 1 on error. By expanding upon this initial knowledge, EESI infers error specifications.

In practice, the domain knowledge required for EESI is very small, typically consisting of only a single error-only function and a few function error-specifications (Section 5.1). For example, OpenSSL was analyzed using the single error-only function `ERR_put_error` and the single initial error-specification that `malloc` returns 0 (null pointer) on error. Error codes are used in prior work [21]. However, the difference lies in the fact that EESI *expands* on this initial domain knowledge, and is able to find function error-specifications that are not restricted to such error codes. For example, EESI is able to infer error specifications for functions that return zero, positive, or nonnegative values on error even though the error codes in the Linux kernel are negative integers.

Errors may propagate through long function-call chains, often crossing subsystem boundaries. For example, a memory allocation error starting at the Linux kernel slab allocator will be first returned as a null pointer from `slab_alloc` before being converted to a negative error code in the IP routing function `ip_route_input_mc`, before finally being converted to a positive error value in `xfrm4_rcv_encap_finish`, seven function calls away from the original error. A developer might find it difficult to manually infer the function error-specification. Documentation of function error-specifications, if available, is often incorrect [22]. The function error-specifications inferred by EESI can be used at development time to determine what errors need to be handled. Consequently, EESI needs to be scalable. It cannot rely on clients of the code to infer the error specifications, and needs to infer error specifications for all functions in the program, not just public API functions.

EESI casts function error-specification inference as computing the least fixpoint of a set of constraints. The constraints are constructed via a flow-sensitive analysis of the body of functions; the inferred error specifications are context insensitive (Section 3). This formulation enables EESI to scale to large programs, while still maintaining precision. EESI takes 5 min 25 sec to analyze 320K lines of Linux file-system code, and obtains an overall precision of 0.93 (Section 5.2).

This paper also presents ECC (Section 4), an automated tool that uses the function error-specifications inferred by EESI to find error-handling bugs, such as insufficient error checks. ECC detected 246 error-handling bugs across 9 programs, of which 110 have been confirmed. ECC detected 220 previously unknown bugs, of which 99 are confirmed, and we are in the process of confirming 107 potential bugs. Two patches have already been merged into OpenSSL (Section 5.4).

The contributions of this paper can be summarized as:

- We develop EESI, a static analysis to infer function error-specifications using domain knowledge (Section 3).
- We develop ECC, a tool that uses function error-specifications to find error-handling bugs (Section 4).

```
1  static int ext4_dx_csum_verify(struct inode *inode,
2           struct ext4_dir_entry *dirent)  {
3    struct dx_countlimit *c;
4    struct dx_tail *t;
5    int count_offset, limit, count;
6
7    if (!ext4_has_metadata_csum(inode->i_sb))
8      return 1;
9
10   c = get_dx_countlimit(inode,dirent,&count_offset);
11   if (!c) {
12     EXT4_ERROR_INODE(inode, "dir seems corrupt?  Run
         e2fsck -D.");
13     return 0;
14   }
15   limit = le16_to_cpu(c->limit);
16   count = le16_to_cpu(c->count);
17   if (count_offset+(limit*sizeof(struct dx_entry)) >
        EXT4_BLOCK_SIZE(inode->i_sb) - sizeof(struct
        dx_tail)) {
18     warn_no_space_for_csum(inode);
19     return 0;
20   }
21   t = (struct dx_tail*)(((struct dx_entry*)c)+limit);
22
23   if (t->dt_csum != ext4_dx_csum(inode, dirent,
24       count_offset, count, t))
25     return 0;
26   return 1;
27 }
```

Figure 1: The function `ext4_dx_csum_verify` returns 1 for success and 0 for error.

- We evaluate the precision of the error specifications inferred by EESI on real-world C code (Section 5.2), and compare EESI with the state of the art (Section 5.3).
- We evaluate the effectiveness of ECC at finding error-handling bugs in real-world C code (Section 5.4).

2 OVERVIEW

Figure 1 shows the `ext4_dx_csum_verify` function from the ext4 Linux file system. Similar to the function `f1` we discussed in Section 1, it can return either `0` or 1. EESI is bootstrapped with the initial domain knowledge that the function `EXT4_ERROR_INODE` is an *error-only function*: it is only called on error paths. Because `ext4_dx_csum_verify` must return `0` after the call to `EXT4_ERROR_INODE` on Line 13, EESI infers that `0` is an error value for `ext4_dx_csum_verify`.

This example illustrates some of the challenges of inferring function error-specifications. One challenge is that the error specification of a function cannot be inferred from its return type. Pointer-returning functions often return a null pointer as an error value, but not always. Identifying such a default error value is even more difficult for integer-returning functions, such as `ext4_dx_csum_verify`. Some programs recommend a specific convention for error values, but these are not always strictly adhered to. For instance, although many Linux functions return `0` on success and a negative error code on failure [21], the function `ext4_dx_csum_verify` is one of many examples that do not follow this convention. The error specification of `ext4_dx_csum_verify` is undocumented, and

```
1  static struct buffer_head *__ext4_read_dirblock(...) {
2    if (ext4_dx_csum_verify(inode, dirent))
3      set_buffer_verified(bh);
4    else {
5      ext4_error_inode(inode, func, line, block,
6        "Directory index failed csum");
7      brelse(bh);
8      return ERR_PTR(-EFSBADCRC);
9    }
10 }
```

Figure 2: Excerpt from the function `__ext4_read_dirblock`, which has the only call to `ext4_dx_csum_verify`.

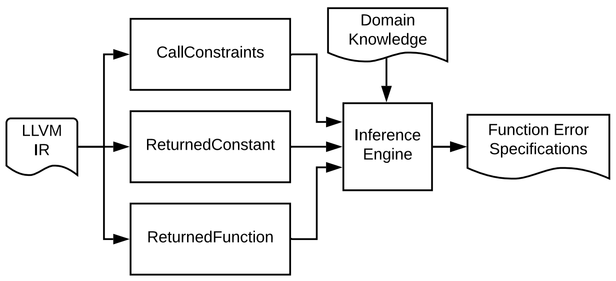

Figure 3: EESI Architecture

a developer must read the definition of the function to learn this specification, often requiring following long call chains.

Prior work [13] deduces error specification of a function from its usage and on empirical characteristics of error paths. However, such an approach must necessarily resolve inconsistencies between call sites through a voting mechanism, and does not work for functions with only a few call sites. For example, the function `ext4_dx_csum_verify` is called exactly once in the Linux kernel, from the function `__ext4_read_dirblock` (Figure 2).

Bugs related to incorrect error handling can be subtle. When this call to `ext4_dx_csum_verify` was introduced into `__ext4_read_dirblock` in February 2013 [28], `ext4_dx_csum_verify` returned 1 in two failure cases, leading to undetected checksum failures in `ext4`, the default file system of many Linux distributions. This issue was fixed in 2016 [12], when `ext4_dx_csum_verify` was patched to always return 0 on failure and 1 on success. This example illustrates the need for an automated technique that infers function error-specifications, and identifies bugs related to incorrect error handling.

3 ERROR-SPECIFICATION INFERENCE

This section describes the static analysis used by EESI to infer function error-specifications (Figure 3) starting with some basic definitions.

Definition 1. An *error value* for a function f is a value returned by f that indicates f encountered a runtime error. ■

Example 3.1. The value 0 is an error value for function `ext4_dx_csum_verify` in Figure 1; this function returns 0 when it encounters a runtime error, for example, on Line 13.

$$\begin{tikzcd}
 & \top \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\leq \mathbf{0} \arrow[d, dash] \arrow[dr, dash] & \neq \mathbf{0} \arrow[dl, dash] \arrow[dr, dash] & \geq \mathbf{0} \arrow[dl, dash] \arrow[d, dash] \\
< \mathbf{0} \arrow[dr, dash] & \mathbf{0} \arrow[d, dash] & > \mathbf{0} \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

$$\gamma(\top) = \{\ldots, -1, 0, 1, \ldots\}$$
$$\gamma(\neq\mathbf{0}) = \{\ldots, -2, -1, 1, 2, \ldots\}$$
$$\gamma(\leq\mathbf{0}) = \{\ldots, -1, 0\} \quad \gamma(<\mathbf{0}) = \{\ldots, -2, -1\}$$
$$\gamma(\geq\mathbf{0}) = \{0, 1, \ldots\} \quad \gamma(>\mathbf{0}) = \{1, 2, \ldots\}$$
$$\gamma(\mathbf{0}) = \{0\} \quad \gamma(\bot) = \emptyset$$
$$\beta(v) = \begin{cases} <\mathbf{0} & \text{if } v < 0 \\ \mathbf{0} & \text{if } v = 0 \\ >\mathbf{0} & \text{if } v > 0 \end{cases}$$

Figure 4: Extended-sign lattice $\mathcal{L}$ with its concretization function $\gamma\colon \mathcal{L} \to \mathcal{P}(\mathbb{Z})$ and abstraction function $\beta\colon \mathbb{Z} \to \mathcal{L}$

The set of error values for a function is its *error specification.* Functions that have no error values, such as the C library function `strcmp`, are called *infallible functions.* However, individual callers of a function might treat certain return values as indicating an error has occurred, but these return values are not error values for the called function. For example, a caller of `strcmp` might treat the result of a specific string comparison as an error, but this does not mean that `strcmp` is encountering an error.

To make the problem of inferring the set of error values tractable, we abstract sets of error values to elements in the *extended-sign lattice* $\mathcal{L}$. The lattice along with its concretization function $\gamma\colon \mathcal{L} \to \mathcal{P}(\mathbb{Z})$ and abstraction function $\beta\colon \mathbb{Z} \to \mathcal{L}$ is shown in Figure 4. For example, the element $\top$ concretizes to the set of all integers $\mathbb{Z}$; $\bot$ concretizes to the empty set; $\beta(-1) = <\mathbf{0}$; $\beta(1) = >\mathbf{0}$; $\beta(0) = \mathbf{0}$. The elements of $\mathcal{L}$ represent the most common checks that developers perform on error values. This lattice captures null-dereference checks because the null value for pointers is represented by 0.

Definition 2. Given the set of functions F in the program, the *error specification* $\mathcal{E} : F \to \mathcal{L}$ maps each function $f \in F$ to $\ell \in \mathcal{L}$ such that $\gamma(\ell)$ contains the set of error values for f. ■

Example 3.2. In Figure 1, $\mathcal{E}($`ext4_dx_csum_verify`$) = \mathbf{0} \in \mathcal{L}$. This function returns 0 on error and there are no other error values.

3.1 Intraprocedural Analyses

This section describes the flow-sensitive, intraprocedural analyses used by EESI. We use F to denote the set of functions in the program, S_f to denote the set of statements in function $f \in F$, and callsites(f) to denote the set of statements that contain a call to function f.

CallConstraints Analysis. *CallConstraints* analysis determines the constraints on function return values necessary to execute a statement.

Definition 3. Given a function $f \in F$, Constraint$_f(s, f') = \ell \in \mathcal{L}$ if the statement $s \in S_f$ may be executed when any call to function $f' \in F$ in f returns a value in $\gamma(\ell)$. ■

Example 3.3. Constraint$_f(s_3,$ `ext4_dx_csum_verify`$) = \neq\mathbf{0} \in \mathcal{L}$ in Figure 2, because the statement on Line 3 (s_3) is executed when the call to the function `ext4_dx_csum_verify` in function $f =$ `__ext4_read_dirblock` returns a non-zero value.

$$\frac{f \in F_{init}}{\mathcal{E}(f) \leftarrow \mathcal{E}_{init}(f)}\ \textsc{InitSpec} \qquad \frac{f \notin F_{init}}{\mathcal{E}(f) \leftarrow \bot}\ \textsc{InitBot} \qquad \frac{s \in S_f \quad \text{RetConst}_f(s) = c \quad c \in EC}{\mathcal{E}(f) \leftarrow \mathcal{E}(f) \sqcup \beta(c)}\ \textsc{ErrorCode}$$

$$\frac{s \in S_f \quad \text{RetConst}_f(s) = c \quad s \in \text{callsites}(f') \quad f' \in F_{eo}}{\mathcal{E}(f) \leftarrow \mathcal{E}(f) \sqcup \beta(c)}\ \textsc{ErrorOnlyCall}$$

$$\frac{s \in S_f \quad \text{RetConst}_f(s) = c \quad \text{Constraint}_f(s, f') = \ell \quad \ell \sqcap \mathcal{E}(f') \neq \bot \quad \ell \neq \top}{\mathcal{E}(f) \leftarrow \mathcal{E}(f) \sqcup \beta(c)}\ \textsc{ErrorConstant}$$

$$\frac{s \in S_f \quad \text{RetFunc}_f(s) = g \quad \text{Constraint}_f(s, f') = \ell \quad \ell \sqcap \mathcal{E}(f') \neq \bot}{\mathcal{E}(f) \leftarrow \mathcal{E}(f) \sqcup \mathcal{E}(g)}\ \textsc{CallPropagation}$$

Figure 5: EESI Inference Rules

ReturnedConstant Analysis. *ReturnedConstant* analysis determines the constant, if any, that must be returned if a statement executes.

Definition 4. Given a function $f \in F$, $\text{RetConst}_f(s) = c \in \mathbb{Z}$ if f must return the constant c if the statement $s \in S_f$ is executed. ■

Example 3.4. $\text{RetConst}_f(s_5) = \text{RetConst}_f(s_7) = \text{RetConst}_f(s_8) =$ -EFSBADCRC in Figure 2, where s_5, s_7, and s_8 are statements on Lines 5, 7, and 8, respectively, in function f = __ext4_read_dirblock, and the macro EFSBADCRC defines a constant.

ReturnedFunction Analysis. *ReturnedFunction* analysis determines the call return value, if any, that must be returned by a function if a statement executes.

Definition 5. Given a function $f \in F$, $\text{RetFunc}_f(s) = f' \in F$ if f must return the value returned by a call to function f' in f if the statement $s \in S_f$ is executed. ■

Example 3.5. $\text{RetFunc}_f(s_5)$ = hid_quirks_init in Figure 8(c), where s_5 is the statement on Line 5, and f = hid_init, because if Line 5 is executed then the function hid_init must return the value returned by a call to hid_quirks_init.

3.2 Domain Knowledge

EESI utilizes three types of domain knowledge to bootstrap the error-specification inference:
(i) Error codes $EC \subseteq \mathbb{Z}$ are specific constants that are used to denote an error value by convention. Consequently, if a function f returns an error code $c \in EC$ then c is an error value for f. For example, macros such as ENOMEM and EFSBADCRC are used to denote error codes in the Linux kernel.
(ii) Error-only functions $F_{eo} \subseteq F$ are functions that are only called when an error has occurred. Consequently, a path in function g has to return an error value if a call to a function $f_{eo} \in F_{eo}$ occurs along that path. For example, ext4_error_inode in Figure 2 is an error-only function. Consequently, -EFSBADCRC is an error value for the function __ext4_read_dirblock.
(iii) Initial error-specification $\mathcal{E}_{init}\colon F_{init} \rightarrow \mathcal{L}$ specifies function error-specifications for the functions $F_{init} \subseteq F$. For example, F_{init} = {malloc} and $\mathcal{E}_{init}$(malloc) $= \mathbf{0} \in \mathcal{L}$ for OpenSSL, which states that malloc returns 0 (null pointer) on error.

3.3 Interprocedural Inference Engine

Figure 5 shows the inference rules that EESI uses to infer the function error-specification $\mathcal{E}\colon F \rightarrow \mathcal{L}$ (Definition 2) using the results of the prior intraprocedural analyses (Section 3.1) as well as the domain knowledge (Section 3.2).

The **InitSpec rule** initializes $\mathcal{E}(f)$ using the initial error specification $\mathcal{E}_{init}(f)$ when $f \in F_{init}$, and the **InitBot rule** initializes $\mathcal{E}(f)$ to $\bot$ when $f \notin F_{init}$.

A constant c is returned by a function $f \in F$ if there exists a statement $s \in S_f$ and $\text{RetConst}_f(s) = c$. The ErrorCode, ErrorOnlyCall, and ErrorConstant rules all determine whether a constant c that can be returned by a function f is an error value for f. If c is determined to be an error value for f, then the error-specification of f is updated using the abstraction of c; that is, $\mathcal{E}(f) \leftarrow \mathcal{E}(f) \sqcup \beta(c)$.

The **ErrorCode rule** states that if the function f can return an error code $c \in EC$, then c is an error value for f.

The **ErrorOnlyCall rule** states that if statement s in function f is a call to an error-only function f_{eo} and the function f returns the constant c when s is executed, then c is an error value for f.

The **ErrorConstant rule** states that if function f returns the constant c when a call to function f' returns an error value, then c is an error value for f. If the condition $\text{Constraint}_f(s, f') = \ell \wedge \ell \sqcap \mathcal{E}(f') \neq \bot$ is true then s in f could be executed when the f' returns an error value. The restriction $\ell \neq \top$ is added to this rule to limit the detrimental impact of missed error checks, or otherwise incorrect code, on specification inference.

The **CallPropagation rule** states that if function f returns the return value of function g when a call to function f' returns an error value, then the error values of g are also error values for f. Hence, the error specification $\mathcal{E}(f)$ can be updated to include the error specification $\mathcal{E}(g)$.

After initializing the analysis using the InitSpec and InitBot rules, the remaining rules are applied until fixpoint, following a standard Kleene iteration sequence. The analysis terminates because the height of the lattice $\mathcal{L}$ is finite. The soundness of the inference rules follows from the soundness of the underlying intraprocedural analyses and the correctness of the domain knowledge.

4 ERROR-HANDLING BUG DETECTION

This section describes a static analysis that finds error-handling bugs using the function error-specification $\mathcal{E}: F \to \mathcal{L}$ inferred by EESI. Specifically, we describe *Error-Check Checker* (ECC) that finds bugs related to insufficient or incorrect error checks.

Definition 6. An *error check* is a conditional branch statement that tests if the value returned by a function call is an error value. ■

ECC finds inconsistencies between the error-specification of a function (Definition 2) and the error checks (Definition 6) associated with calls to that function. These inconsistencies are manifested in three different bug patterns.

(a) Insufficient error checks. Insufficient error checks occur when the error checks associated with a call to a function f fail to cover all of the error values that f may return. This can occur when the return value of a function is not saved at all, saved but not checked, or when the return value of a function is checked for a range of values that is a proper subset of the error values that a function may return.

(b) Inverted error checks. An inverted error check is an error check that gets the direction of the error path wrong. A common cause of inverted error checks is the use of error values that do not conform to the idiomatic error handling conventions used to signal errors in C, such as returning 0 on error. The bug involving the original version of `ext4_dx_csum_verify` described in Section 2 is an instance of an inverted error check. In this case, the bug was fixed by modifying `ext4_dx_csum_verify` to return 1 on error.

(c) Incomplete error-specifications. Another type of inconsistency arises when the error check for a function call is correct, but the implementation of the function does not return the correct set of error values. This arises when there are error checks for values that a function cannot return, resulting in dead code. An example of a previously unknown bug of this type that we found in the Linux kernel is shown in Figure 8(b).

5 EXPERIMENTAL EVALUATION

The experiments described in this section were designed to answer the following research questions:

RQ1 How accurately does EESI infer function error-specifications?
RQ2 How does EESI compare with the state of the art?
RQ3 How effective is ECC at finding error-handling bugs when using the function error-specifications inferred by EESI?

5.1 Experimental Setup

Benchmarks. Table 1 lists the programs used in the evaluation of EESI and their size. These programs were chosen to be a representative cross section of important software written in C ranging from operating systems to cryptographic libraries. "Linux FS" stands for Linux File System, which includes the virtual file system (VFS), the Linux memory manager, and four file systems: `ext2`, `ext4`, `btrfs`, and `FAT`. "Linux NFC" is the near-field communication subsystem of the Linux kernel. "Full Linux kernel" refers to a runnable Linux kernel including all components in the default configuration.

Domain Knowledge. We used the following types of domain knowledge (see Section 3.2 and Table 2) when running EESI:

(i) Error codes (EC). 34 error codes were used when running EESI on Linux FS, Linux NFC, and Full Linux kernel. No error codes were used for the rest of the programs.
(ii) Error-only functions. EESI required the use of few error-only functions. Finding these error-only functions was easy, because they mostly contained `error` in their name. For example, OpenSSL was analyzed using the single error-only function `ERR_put_error`.
(iii) Initial error-specification. EESI required the use of few initial error-specifications. The error specification for allocation functions, such as `malloc`, `calloc`, and `__slab_alloc`, was used as initial error-specification when running EESI on the programs, as listed in Table 2. Error specifications for 13 `pthread` library functions was used as initial error-specification when analyzing `netdata`.

The error specifications for the following functions was also used when analyzing Linux FS: `sync_inode_metadata`, `ext4_inode_loc` and `jbd2_journal_metadata`. These functions were identified by looking at the position of their corresponding vertices in the return propagation graph generated for Linux file systems. A *return propagation graph* is a directed graph where every vertex represents a program function and there is an edge from u to v if the function corresponding to v propagates the return value of the function corresponding to u. If many vertices are reachable from a source vertex u in the return propagation graph, then EESI is more likely to infer error specifications for new functions when the error specification for the function corresponding to u is provided as an initial error-specification.

Analysis and Bug Checking Performance. EESI and ECC are implemented using LLVM [14], and are available at https://github.com/ucd-plse/eesi. The full Linux kernel analysis was run on an Amazon EC2 r4.2xlarge instance, while the rest were run on a 3.60 GHz i7-4790 CPU with 32 GB of RAM. Table 1 shows the runtime performance of EESI and ECC. For 6 out of 9 programs, EESI takes less than one minute to run. The analyses of OpenSSL, Linux FS, and Full Linux kernel take 1 min 33 sec, 5 min 25 sec, and 12 min 47 sec, respectively. ECC is also efficient, with Full Linux kernel taking the most time, 11 min 34 sec.

5.2 RQ1: Accuracy of EESI

EESI identifies error specifications for 18,919 functions among all programs analyzed. Table 2 provides an overview of the types of

Table 1: Size of programs in KLOC (thousands of lines of code), and runtime performance of EESI and ECC (times are minutes:seconds of elapsed wall clock time).

	KLOC	EESI	ECC
OpenSSL	231	1:33	2:48
Pidgin OTRv4	7	0:05	0:04
mbedTLS	192	0:32	0:38
netdata	60	0:36	0:59
Linux FS	320	5:25	3:35
Linux NFC	32	0:37	0:35
Full Linux kernel	1,295	12:47	11:34
LittleFS	2	0:03	0:02
zlib	1	0:09	0:08

Table 2: Domain knowledge used by EESI and the total number of specification per type inferred by EESI

		Error-Only Functions		Initial Error-Specifications		**Specifications Inferred by EESI**					
Program	EC	# Funcs	Example	#Specs	Example	<0	0	≤ 0	>0	$\neq 0$	≥ 0
OpenSSL	0	1	`ERR_put_error`	1	`malloc`$(=0)$	268	2,931	259	72	10	173
Pidgin OTRv4	0	0	NA	4	`malloc` $(=0)$	0	12	0	3	0	0
mbedTLS	0	1	`mbedtls_strerror`	2	`calloc` $(=0)$	231	23	22	2	1	0
netdata	0	1	`perror`	16	`malloc` $(=0)$	11	24	1	4	0	14
Linux FS	34	12	`ext4_error`	7	`ext4_inode_loc` (<0)	2,374	668	625	34	7	21
Linux NFC	34	0	NA	4	`__slab_alloc` $(=0)$	838	141	26	18	0	7
Full Linux kernel	34	0	NA	4	`__slab_alloc` $(=0)$	5,861	2,981	578	394	13	163
LittleFS	0	0	NA	1	NA	14	2	28	0	0	0
zlib	0	0	NA	1	`malloc` $(=0)$	53	12	0	0	0	0
Total		15		35		9,650	6,794	1,539	527	31	378

inferred specifications. The two most common types are $<\mathbf{0}$ and $\mathbf{0}$ with a total of 16,444 specifications. We observe that integer-returning functions commonly return a negative number on error, which is a strong convention in the Linux kernel, and that pointer-returning functions commonly return 0 (null pointer) on error. However, these conventions are not always adhered to: EESI also infers 2,475 error specifications of the types $\leq \mathbf{0}$, $>\mathbf{0}$, $\neq \mathbf{0}$, and $\geq \mathbf{0}$. OpenSSL, Linux FS and Full Linux kernel include specifications from all types. For example, EESI infers that the Linux FS function `__ext4_read_dirblock` (Figure 2) has the error specification $<\mathbf{0}$.

We compared the output of EESI with the ground truth for 395 functions to evaluate the accuracy of the specifications inferred by EESI. Ground truth was obtained via manual review of the source code. These 395 specifications included a random sample of 100 (93 correct) specifications from the projects in Table 2, a random sample of 50 (47 correct) OpenSSL functions, all 95 (92 correct) functions defined by the zlib library, and an additional 150 (137 correct) randomly sampled specifications in OpenSSL that overlap between EESI and APEx (Section 5.3.2). In total, 369 of the 395 function error-specifications inferred by EESI exactly matched the ground truth. Thus the estimated precision of EESI is 0.93.

One of the primary sources of inaccuracy in EESI is when a function uses an out parameter instead of a return value to signal errors. Figure 6 shows one such case where the value 0 is incorrectly inferred as an error value for `tcp_fastopen_defer_connect`, when actually the pointer argument `*err` is set to `-ENOBUFS`. Pointer operands behave similarly to return values in that the caller needs to check `*err` instead of the return value. We plan to address this in future versions of the EESI implementation.

```c
bool tcp_fastopen_defer_connect(struct sock *sk,
        int *err) {
  ...
  if (tp->fastopen_req)
     tp->fastopen_req->cookie = cookie;
  else
     *err = -ENOBUFS;
  return false; // false is defined as 0
}
```

Figure 6: EESI incorrectly infers that 0 is an error value for `tcp_fastopen_defer_connect` due to the use of an out parameter `err` to signal errors.

From manual inspection of 395 functions, **we conclude that EESI infers error specifications with a precision of 0.93**, answering RQ1.

5.3 RQ2: Comparison with State-of-the-Art

This section presents a qualitative (Table 3) and quantitative (Table 4) comparison of EESI with the error-specification inference tool APEx [13].

5.3.1 Qualitative Comparison. We compare the tools in terms of the following four characteristics:

(1) EESI analyzes implementations directly. EESI infers function error-specifications directly from the program. APEx relies on clients of the program to infer function error-specifications. Obtaining and building individual clients adds a significant amount of manual effort to the specification inference task.

(2) EESI infers specifications for internal and API functions. EESI infers specifications for internal and API functions, while APEx is limited to frequently used API functions. An API function

Table 3: Qualitative comparison of EESI with APEx. (1) EESI analyzes programs directly instead of requiring their clients. (2) EESI infers function error specifications for both internal and API functions. (3) EESI incorporates domain knowledge, instead of relying on the path-length heuristic used by APEx. (4) EESI scales to large programs, while APEx does not due to its use of path-sensitive symbolic execution.

	EESI	APEx
(1) Direct Analysis	✓	✗
(2a) API Functions	✓	✓
(2b) Internal Functions	✓	✗
(3) Domain Knowledge	✓	✗
(4) Scalable	✓	✗

Table 4: Quantitative comparison of EESI with APEx. *AllFns* is the total number of non-void functions defined by the library. *APIFns* is the number of non-void functions defined by the library that are called from one of the clients listed in Section 5.3.2. *Time* compares the time each tool requires to perform its analysis (mm:ss). *Total Specs* is the intersection of the tool output with *AllFns*. *API Specs* is the intersection of the tool output with *APIFns*. *Precision* is the ratio of correct specifications reported by the tool to the total number of API specs reported by the tool. *Recall* is the ratio of the total number of API specs reported by the tool to the total number of API specs that can return an error. The OpenSSL results were calculated for a random sample of 50 API functions.

			Time		Total Specs		API Specs		Precision		Recall	
	AllFns	APIFns	EESI	APEx	EESI	APEx	EESI	APEx	EESI	APEx	EESI	APEx
zlib	147	25	0:15	121:15	65	8	17	8	1.0	0.75	0.71	0.33
OpenSSL	7,031	644	1:19	93:20	3,713	313	339	313	0.88	0.76	0.68	0.45

is a function that is defined in a library, and called by clients of the library. Internal functions are functions that are not available to clients. Internal functions are often refactored into small functions which are only called from a few locations. Therefore, relying on patterns among a large number of calls to these functions to determine the error specification of a function is not a viable option.

(3) EESI incorporates domain knowledge. EESI is bootstrapped with small amounts of developer-provided knowledge. This input provides a firm foundation on which additional function error specifications can be inferred. In contrast, APEx relies on the assumption that error paths are shorter than non-error paths. This assumption frequently does not hold. The function `sidtab_reverse_lookup` is one example that illustrates the problem with this approach. An excerpt from this function, simplified for the purpose of presentation, is shown in Figure 7. At the top of the function, a cache lookup is performed to check if the entry can be returned. When the cache lookup is successful, the function returns `0` along the shortest path through the function. The error code `-ENOMEM` is returned on failure; this error path is considerably longer than the success path.

(4) EESI scales to large programs. Even for large programs such as the Linux kernel, EESI is able to infer error specifications for thousands of functions (Table 2) in only a few minutes (Table 1). APEx can take hours to infer specifications for only the API functions in smaller user-space libraries [13]. APEx relies on path-sensitive symbolic execution provided by clang static analyzer, which has already been extensively optimized for performance.

```
static int sidtab_reverse_lookup(struct sidtab *s,
    struct context *context, u32 *index) {
  ...
  rc = sidtab_rcache_search(s, context, index);
  if (rc == 0)
    return 0;
  rc = -ENOMEM;
  ... // 35 statements omitted
  rc = 0;
out_unlock:
  return rc;
}
```

Figure 7: Example of a short non-error path from Linux

5.3.2 *Quantitative Comparison.*

We provide a quantitative comparison of EESI with APEx for the libraries OpenSSL and zlib. We used the clients listed in [13, Table 5] to infer specifications for OpenSSL and zlib with APEx: clamav-0.101.2, curl-7.64.1, gnutls-3.6.7, httpd-1.4.53, lighttpd-1.4.53, lynx-2.8.9, nginx-1.15.12, openssh-8.0p1 tor-0.3.5.8, and mutt-1.12.1. We compared APEx results to the results obtained when running EESI on OpenSSL and zlib directly (without the need to analyze their clients).

Note that APEx can treat an application as a client of itself, however this requires the application to include a large number of calls to its functions. We attempted to use APEx to infer specifications in the Linux kernel when treating it as a client of itself. APEx crashed when run on the Linux kernel; the symbolic execution phase produced constraints that the APEx analysis scripts were unable to process. APEx also crashed when using OpenSSL as a client of itself, thus we did not include it when running APEx for OpenSSL. We did not consider running APEx on the rest of our programs because either they are not libraries, or they do not include a large number of function calls for APEx to be effective. To gather run-time performance, the tools were run on an Amazon EC2 c5.9xlarge instance with 36 CPU cores and 72GB of memory.

All 25 non-void zlib API functions were used for evaluation. Due to the size of the OpenSSL library, a random sample of 50 non-void OpenSSL API functions were used for evaluation. Precision and recall are defined in terms of the expected number of API functions; the number of API functions that are called from any of the clients considered by APEx that can return an error value according to ground truth. Ground truth was established by manually reviewing the zlib and OpenSSL source code. Of the 25 zlib API functions, 24 could return an error. Of the 50 randomly sampled OpenSSL functions, 37 could return an error.

Definition 7. *Precision* and *Recall* are defined as:

$$Precision \overset{\text{def}}{=} \frac{|Correct|}{|Total|} \qquad Recall \overset{\text{def}}{=} \frac{|Total \cap Ground|}{|Ground|}$$

where $Total$ is the set of specifications reported by the tool, *Correct* is the set of specifications reported by the tool that match the ground truth, and *Ground* is the ground truth set of non-void functions that can return an error. ■

Table 4 lists the total number of functions and the total number of API functions for each library. The table also summarizes

Table 5: Summary of bugs reported by ECC.
TBR: total bug reports, CT: confidence threshold, IBR: inspected bug reports, CB: confirmed bugs, PB: potential bugs, BB: benign bugs, FP: false positives.

				IBR Breakdown			
	TBR	CT	IBR	CB	PB	BB	FP
OpenSSL	2,014	0.8	112	15	48	2	47
Pidgin OTRv4	43	0	43	31	7	0	5
mbedTLS	6	0	6	1	0	1	4
netdata	58	0	58	35	15	0	8
Linux FS	1,470	0.8	49	7	12	5	25
Linux NFC	242	0.8	29	2	9	11	7
Full Linux kernel	2,873	0.9	53	19	13	4	17
LittleFS	2	0	2	0	0	0	2
zlib	15	0	15	0	3	6	6
Total	6,723		367	110	107	29	121

the results. Because APEx relies on clients to infer function error-specifications, it only infers specifications for API functions and not internal functions. This is reflected in the difference in the total number of specifications inferred by each tool (the *Total Specs* column in Table 4). In particular, EESI finds 8× and 11× more specifications than APEx in zlib and OpenSSL, respectively. When restricting the comparison to API functions, EESI also finds more specifications than APEx while exhibiting higher precision and recall for both libraries in considerably less time. EESI took 15 sec to analyze zlib, and 1 min 19 sec to analyze OpenSSL. APEx took 2 hours and 1.5 hours to analyze the clients of zlib and OpenSSL, respectively.

In addition to the 50 randomly sampled OpenSSL API functions, we also randomly sampled 150 OpenSSL specifications for functions where both EESI and APEx provided a specification. Of these, 137 EESI specifications and 118 APEx specifications were correct.

EESI infers specifications for internal and API functions, is not dependent on the path-length heuristic, is more scalable, is more precise, and infers more API function specifications than the state of the art. This answers RQ2.

5.4 RQ3: Usefulness of EESI Specifications in Bug Finding

In this section, we evaluate the bug-finding effectiveness of ECC (Section 4), which uses the function error-specifications inferred by EESI (Section 3). Table 5 summarizes the bug reports produced by ECC. The total bug reports (TBR) per program varied from 2 to 2,873. The IBR column lists the number of bug reports we manually inspected. We inspected all bug reports for the 5 programs for which ECC generated less than 100 bug reports. For the remaining 4 programs, we computed a *confidence* for each bug report, and only inspected bug reports whose confidence was greater than or equal to the *confidence threshold (CT)*. The confidence threshold was chosen so as to limit the number of inspected bug reports to around 100 per program. The *confidence* of a bug report involving the return value of a function f is defined as the number of calls to f that have correct error-checks divided by the total number of calls to f. The rest of the section describes the breakdown of the inspected bug reports (IBR Breakdown).

Confirmed and potential bugs. In total, we found *110 confirmed bugs (CB)* and *107 potential bugs (PB). Confirmed bugs* include the following: bugs that we reported and were confirmed by developers (2), bugs that were independently found by others (11), and bugs that we confirmed ourselves (97). *Potential bugs* are instances in which the report is not an obvious false positive, but the complexity of the code prevents us from confirming the bug without additional input from developers. *Of the 110 confirmed bugs, 99 bugs were previously unknown; all of the 107 potential bugs were previously unknown.* Here previously unknown means that, to our knowledge, no one knew about them. All bugs were previously unknown to us. Note that confirmed bugs have been found in all programs except for LittleFS and zlib. We are in the process of reporting all confirmed and potential bugs to developers. Patches we provided for two of the OpenSSL bugs were merged into OpenSSL for the 1.1.1b release [5, 6].

Confirmed bugs in Linux. ECC found *24 previously-unknown, confirmed bugs* in version 5.0-rc3 of the Linux kernel using error specifications generated by EESI. Figure 8 shows one such bug. In Figure 8(a), the function `hid_modify_dquirk` returns the error code `-ENOMEM` on Line 6 if `kzalloc` is unable to allocate memory. The function `hid_quirks_init` in Figure 8(b) correctly checks the return value of `hid_modify_dquirk` on Line 5, but fails to propagate the error value. Consequently, EESI infers that the error specification for `hid_quirks_init` is $\bot$. Not all errors need to be propagated, but in this case we observe an inconsistency between the error specification for `hid_quirks_init` and the error check on Line 4 in Figure 8(c). The error check on Line 4 results in dead code.

Confirmed bugs in OpenSSL. ECC found *8 previously-unknown, confirmed bugs* in version 1.1.1a of OpenSSL. Patches sent by us for two of these bugs have been merged [5, 6], and we are in the process of reporting the remaining.

Figure 9 shows a previously unknown bug that ECC found in OpenSSL. On Line 6, the call to `M_ASN1_new_of` can return a null pointer on memory-allocation failure. On Line 9, this pointer is dereferenced, resulting in a segmentation fault if `rek` is `null`. We generated a patch for this bug, which was accepted by the OpenSSL developers and merged into OpenSSL 1.1.1b [6].

It is not immediately obvious that `M_ASN1_new_of` can return a null pointer at all, and even less obvious that it returns a null pointer in response to memory-allocation failure. `M_ASN1_new_of` is a macro wrapping the function `ASN1_item_new`. `ASN1_item_new` returns null when `ASN1_item_ex_new` returns null, which in turn propagates the error from `asn1_item_embed_new`, which returns null when the `OPENSSL_zalloc` macro wrapping `CRYPTO_zalloc` fails, which propagates errors from `CRYPTO_malloc`. Finally, `CRYPTO_malloc` fails when `malloc` returns a null pointer, allowing EESI to infer that on Line 6, `rek` will be null when `malloc` fails. Tracking down such long error-propagation chains makes it difficult for a developer to manually infer function error specifications.

Because of its importance [18], OpenSSL has been reviewed extensively. In January 2019, Quarkslab performed a security assessment of the OpenSSL code, spending 60 man-days to audit four

```c
static int hid_modify_dquirk(...) {
 ...
 int ret = 0;
 hdev = kzalloc(sizeof(*hdev),...);
 if (!hdev)
   return -ENOMEM;
 ...
out:
 kfree(hdev);
 return ret;
}
```

(a) `hid_modify_dquirk` returns zero on success and a negative error on failure.

```c
int hid_quirks_init(...) {
 ...
 for (;n<count && qparam[n]; n++) {
  ...
  if (hid_modify_dquirk(...) != 0){
   pr_warn("Could not parse HID
        quirk module param");
  }
 }
 return 0;
}
```

(b) `hid_quirks_init` checks for `hid_modify_dquirk`'s error, but fails to propagate it.

```c
static int __init hid_init(void) {
  int retval = -ENOMEM;
  retval = hid_quirks_init(...);
  if (retval)
    goto usbhid_quirks_init_fail;
  ...
  return 0;
  ...
usbhid_quirks_init_fail:
  return retval;
}
```

(c) `hid_init` expects error to propagate, resulting in dead code.

Figure 8: Bug found by ECC in Linux 5.0-rc3 resulting from an incomplete error specification. The error originating in (a) is not propagated by the function shown in (b). The missing propagation results in dead code in (c).

```c
int cms_RecipientInfo_kari_init(...) {
  ri->d.kari=M_ASN1_new_of(CMS_KeyAgreeRecipientInfo);
  if (!ri->d.kari)
    return 0;
  ...
  rek = M_ASN1_new_of(CMS_RecipientEncryptedKey);
  ...
  if (flags & CMS_USE_KEYID) {
    rek->rid->type = CMS_REK_KEYIDENTIFIER;
  }
}
```

Figure 9: Null pointer dereference found by ECC in OpenSSL, which was previously unknown to the OpenSSL developers. We provided a patch that was merged into OpenSSL 1.1.1b.

components of OpenSSL [1]. In the code-quality section of their report on the Secure Remote Password (SRP) protocol, they find eight cases where the return value of a function is not checked for errors. *Seven of these eight bugs are reported by ECC.*

Confirmed bugs in Pidgin OTRv4. ECC found *31 previously-unknown, confirmed error-handling bugs* in the Pidgin plugin that supports the upcoming v4 standard of Off-the-record messaging (OTR). OTR provides deniability for instant messaging conversations [3], making it useful for journalists and other actors in situations where it might be important to deny that a conversation occurred. This OTR plugin was chosen because of its global importance, and because the developers had identified error-handling bugs as a high priority. The defects identified by ECC would lead to crashes or other undefined behavior in the plugin.

Benign bugs. Table 5 also reports 29 *benign bugs (BB)*. These are instances in which checks are indeed missing (ECC correctly reports them), but the missing checks do not result in a serious enough problem to warrant a fix. An example of this would be an unchecked output-error during logging; even though the specification is correct, the error is considered benign in our evaluation.

False positives. Table 5 shows the number of *false positives* (FP) reported by ECC. Table 6 shows a breakdown of the types of false positives we encountered while inspecting the bug reports for the three programs with the most false positives. "Incorrect Spec" false positives occur in ECC when EESI has inferred the incorrect error

Table 6: Breakdown of false positives in ECC bug reports for a subset of programs.

	Linux FS	OpenSSL	Full Linux	Total
Incorrect Spec	1	0	0	1
Missed Checks	21	29	14	64
Interprocedural	1	8	2	11
Out Parameter	1	10	0	11
`nofail`	1	0	1	2
Total	25	47	17	89

specification for a function. Only one of the false positives in ECC was due to inaccuracies in error specifications inferred by EESI.

The largest number of false positives are due to "Missed Checks", where the implementation of ECC failed to identify an error check. For example, the Linux kernel defines assertion functions that crash the kernel on certain conditions. ECC is not aware of these functions and, therefore, reports the return value as unchecked. Improving ECC to remove such false positives is part of future work.

"Interprocedural" false positives occur when the error value is passed as an argument to callee function that contains the error check. "Out Parameter" false positives occur when the error value is assigned to an out parameter and the caller function contains the error check. Finally, the "`nofail`" false positives are peculiar to Linux, where memory allocations can be requested that will not fail (the memory allocator will loop indefinitely).

ECC detected **246 error-handling bugs** across 9 programs, of which 110 have been confirmed. ECC **detected 220 previously unknown bugs**, of which 99 are confirmed, and we are in the process of confirming 107 potential bugs. Two patches have already been merged into OpenSSL. Finally, ECC identified 29 benign bugs. This answers RQ3.

5.5 Threats to Validity

The function error-specifications inferred by EESI are, of course, dependent upon the domain knowledge provided. As seen in Table 2, the total number of inputs is small, and it would be easy for project developers to provide more domain knowledge with little effort.

EESI was evaluated on the programs in Table 2. Our results may not generalize to software that exhibits significantly different error-handling behavior. However, these programs were chosen to be a representative cross section of important software written in C.

6 RELATED WORK

Error-handling specifications. Acharya and Xie [2] use data mining techniques on static traces to mine error-handling specifications for relevant APIs used in software packages. The approach follows a restricted classification of error-handling code and limits to identifying error checks and cleanup code in client code. Kang et al. [13] introduce APEx, a tool for finding error specifications for API C functions. Section 5.3 presents a detailed qualitative and quantitative comparison between EESI and APEx.

Fault-injection techniques have also been used to extract error-handling specifications [8, 19, 20, 25]. Fetzer et al. [8] introduce the notion of failure atomicity in the context of exceptions and propose techniques to automatically detect and mask non-atomic exception handling in C++ and Java applications. Süßkraut and Fetzer [25] detect and patch incorrect C error-handling client code. Incorrect error-handling code is identified when the system crashes. Patching transforms unhandled errors into errors the application can handle. Prabhakaran et al. [19, 20] build models of how journaling file systems must behave under different journaling modes, and use these to find error-handling specifications related to disk write failures. Marinescu and Candea [16] describe a framework for testing recovery code through error-code injection.

EESI is a static analysis tool and, therefore, better suited to handling systems software such as the Linux kernel. The Linux kernel comes equipped with a fault injection framework, but injecting errors into software that interacts with devices is difficult as it requires the hardware to be present. For user-space code, EESI is complementary to dynamic fault injection.

Function error-specifications to find error-handling bugs. A number of tools require function error-specifications to detect bugs or infer how errors should be handled. EPex [11] takes as input the error values that a function can return and reports as potential bugs error paths that do not handle the error, where error handling is defined as returning an error value, logging, or exiting. ErrDoc [27] is an improvement over EPex, which takes as input specifications inferred using APEx. Because APEx cannot be used to infer specifications for functions in the Linux kernel, ErrDoc cannot be used to find or fix error-handling bugs in the Linux kernel. DeFreez et al. [7] created Func2vec to embed functions in a vector space such that functions that fulfill the same role or purpose are in close proximity, and used this embedding to improve the quality of error-handling specifications. Their specification miner takes as input function error-specifications. EESI is scalable and capable of inferring more function specifications than the state of the art. Therefore, all these tools could benefit from EESI.

Other approaches to finding error-handling bugs. Static analysis techniques [9, 21, 23, 31] have been proposed to track the propagation of error codes in systems software to find a wide range of error-handling bugs such as dropped error codes. Saha et al. [24] present a static analysis to find resource-release omission faults in C code. JUXTA [17] is a symbolic-execution based approach to find semantic bugs (including error-handling bugs) across Linux file systems. Henkel et al. [10] use code embeddings to detect incorrect returned error codes in Linux code. By using the expressive error-specifications inferred by EESI, ECC is able to find subtle error-handling bugs that do not necessarily involve error codes (see Section 2 and Section 4), which is beyond the capability of the above techniques. Finally, a large body of work (e.g., [4, 26, 29, 30]) has proposed static analysis to find error-handling bugs in Java programs, which are outside the scope of this paper.

7 CONCLUSION

This paper presented EESI, a static analysis to infer function error-specifications for programs written in C that use the return-code idiom. EESI bootstraps the analysis by using three types of domain knowledge: error codes, error-only functions, and initial error-specifications. The inference rules used by EESI expand on this initial domain knowledge to infer additional function error-specifications. Our evaluation of EESI on real-world programs, such as OpenSSL and the Linux kernel, show that EESI can accurately infer function error-specifications while scaling to large programs.

We demonstrated how the function error-specifications inferred by EESI can be used to automatically find bugs related to incorrect error handling by building a tool named ECC to find three types of error-handling bugs: insufficient error checks, inverted error checks, and incomplete error specifications. ECC detected 246 bugs across 9 programs, of which 110 have been confirmed as actual bugs. ECC detected 220 previously unknown bugs, of which 99 are confirmed, and we are in the process of confirming 107 potential bugs. Two patches have already been merged into OpenSSL.

The careful orchestration of intraprocedural flow-sensitive analyses and interprocedural context-insensitive analysis allows EESI to be scalable and precise. As shown in our evaluation, EESI takes only minutes to run on even very large programs such as the Linux kernel. EESI outperforms the state of the art [13] in precision, recall, and performance. Furthermore, by not relying on heuristics based on usage of functions or empirical properties of error paths, EESI is more generally applicable.

The scalability of EESI makes it a good fit for continuous integration and delivery pipelines that run tools on every commit. EESI could be used to notify developers when the error specification of a function has changed. Given the long error-propagation chains that occur in large programs, this can be particularly useful, as changing the error specification of a function can have unintended consequences that result in defective error-handling code.

ACKNOWLEDGMENTS

The authors would like to thank Sung Kook Kim, Eric Li, Matthew Sotoudeh, and James Sun for their valuable comments. This material is based upon work supported by the National Science Foundation under Grant No. 1464439 and Grant No. 1750983, a Facebook Probability and Programming research award, and AWS Cloud Credits for Research.

REFERENCES

[1] The ostif and quarkslab audit of openssl is complete (2019), https://ostif.org/the-ostif-and-quarkslab-audit-of-openssl-is-complete/, accessed: 2019-01-30

[2] Acharya, M., Xie, T.: Mining API error-handling specifications from source code. In: Chechik, M., Wirsing, M. (eds.) Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009. Lecture Notes in Computer Science, vol. 5503, pp. 370–384. Springer (2009), http://dx.doi.org/10.1007/978-3-642-00593-0_25

[3] Borisov, N., Goldberg, I., Brewer, E.A.: Off-the-record communication, or, why not to use PGP. In: Atluri, V., Syverson, P.F., di Vimercati, S.D.C. (eds.) Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004. pp. 77–84. ACM (2004), https://doi.org/10.1145/1029179.1029200

[4] Buse, R.P.L., Weimer, W.: Automatic documentation inference for exceptions. In: Ryder, B.G., Zeller, A. (eds.) ISSTA. pp. 273–282. ACM (2008)

[5] DeFreez, D.: Fix null pointer dereference in cms_recipientinfo_kari_init (2019), https://github.com/openssl/openssl/commit/b754a8a1590b8c5c9662c8a0ba49573991488b20, accessed: 2019-06-30

[6] DeFreez, D.: Fix null pointer dereference in ssl_module_init (2019), https://github.com/openssl/openssl/commit/b754a8a1590b8c5c9662c8a0ba49573991488b20, accessed: 2019-06-30

[7] DeFreez, D., Thakur, A.V., Rubio-González, C.: Path-based function embedding and its application to error-handling specification mining. In: [15], pp. 423–433, https://doi.org/10.1145/3236024.3236059

[8] Fetzer, C., Högstedt, K., Felber, P.: Automatic detection and masking of non-atomic exception handling. In: 2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings. pp. 445–454. IEEE Computer Society (2003), https://doi.org/10.1109/DSN.2003.1209955

[9] Gunawi, H.S., Rubio-González, C., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Liblit, B.: EIO: error handling is occasionally correct. In: Baker, M., Riedel, E. (eds.) 6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA. pp. 207–222. USENIX (2008), http://www.usenix.org/events/fast08/tech/gunawi.html

[10] Henkel, J., Lahiri, S.K., Liblit, B., Reps, T.W.: Code vectors: understanding programs through embedded abstracted symbolic traces. In: [15], pp. 163–174, https://doi.org/10.1145/3236024.3236085

[11] Jana, S., Kang, Y.J., Roth, S., Ray, B.: Automatically detecting error handling bugs using error specifications. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 345–362. USENIX Association (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana

[12] Jeong, D.: ext4: correct error value of function verifying dx checksum (2016), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=fa96454069b85a7e5d10f38b7d95edcd5dc64b9a, accessed: 2019-07-04

[13] Kang, Y.J., Ray, B., Jana, S.: Apex: automated inference of error specifications for C apis. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016. pp. 472–482. ACM (2016), http://doi.acm.org/10.1145/2970276.2970354

[14] Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88 (2004), https://doi.org/10.1109/CGO.2004.1281665

[15] Leavens, G.T., Garcia, A., Pasareanu, C.S. (eds.): Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. ACM (2018), http://dl.acm.org/citation.cfm?id=3236024

[16] Marinescu, P.D., Candea, G.: Efficient testing of recovery code using fault injection. ACM Trans. Comput. Syst. 29(4), 11:1–11:38 (2011), https://doi.org/10.1145/2063509.2063511

[17] Min, C., Kashyap, S., Lee, B., Song, C., Kim, T.: Cross-checking semantic correctness: the case of finding file system bugs. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015. pp. 361–377. ACM (2015), https://doi.org/10.1145/2815400.2815422

[18] Nemec, M., Klinec, D., Svenda, P., Sekan, P., Matyas, V.: Measuring popularity of cryptographic libraries in internet-wide scans. In: Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017. pp. 162–175 (2017), https://doi.org/10.1145/3134600.3134612

[19] Prabhakaran, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Model-based failure analysis of journaling file systems. In: 2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings. pp. 802–811. IEEE Computer Society (2005), https://doi.org/10.1109/DSN.2005.65

[20] Prabhakaran, V., Bairavasundaram, L.N., Agrawal, N., Gunawi, H.S., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: IRON file systems. In: Herbert, A., Birman, K.P. (eds.) Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005. pp. 206–220. ACM (2005), https://doi.org/10.1145/1095810.1095830

[21] Rubio-González, C., Gunawi, H.S., Liblit, B., Arpaci-Dusseau, R.H., Arpaci-Dusseau, A.C.: Error propagation analysis for file systems. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 270–280. ACM (2009), https://doi.org/10.1145/1542476.1542506

[22] Rubio-González, C., Liblit, B.: Expect the unexpected: error code mismatches between documentation and the real world. In: Lerner, S., Rountev, A. (eds.) Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010. pp. 73–80. ACM (2010), https://doi.org/10.1145/1806672.1806687

[23] Rubio-González, C., Liblit, B.: Defective error/pointer interactions in the linux kernel. In: Dwyer, M.B., Tip, F. (eds.) Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011. pp. 111–121. ACM (2011), https://doi.org/10.1145/2001420.2001434

[24] Saha, S., Lawall, J., Muller, G.: Finding resource-release omission faults in linux. In: Proceedings of the 6th Workshop on Programming Languages and Operating Systems, PLOS@SOSP 2011, Cascais, Portugal, October 23, 2011. pp. 1:1–1:5. ACM (2011), https://doi.org/10.1145/2039239.2039241

[25] Süßkraut, M., Fetzer, C.: Automatically finding and patching bad error handling. In: Sixth European Dependable Computing Conference, EDCC 2006, Coimbra, Portugal, 18-20 October 2006. pp. 13–22. IEEE Computer Society (2006), https://doi.org/10.1109/EDCC.2006.3

[26] Thummalapenta, S., Xie, T.: Mining exception-handling rules as sequence association rules. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. pp. 496–506. IEEE (2009), https://doi.org/10.1109/ICSE.2009.5070548

[27] Tian, Y., Ray, B.: Automatically diagnosing and repairing error handling bugs in C. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017. pp. 752–762. ACM (2017), https://doi.org/10.1145/3106237.3106300

[28] Ts'o, T.: ext4: refactor code to read directory blocks into ext4_read_dirblock() (2013), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=dc6982ff4db1f47da73b1967ef5302d6721e5b95, accessed: 2019-07-04

[29] Weimer, W., Necula, G.C.: Finding and preventing run-time error handling mistakes. In: Vlissides, J.M., Schmidt, D.C. (eds.) Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada. pp. 419–431. ACM (2004), https://doi.org/10.1145/1028976.1029011

[30] Weimer, W., Necula, G.C.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3440, pp. 461–476. Springer (2005), https://doi.org/10.1007/978-3-540-31980-1_30

[31] Weiss, C., Rubio-González, C., Liblit, B.: Database-backed program analysis for scalable error propagation. In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1. pp. 586–597. IEEE Computer Society (2015), https://doi.org/10.1109/ICSE.2015.75