APISan: Sanitizing API Usages through Semantic Cross-Checking

Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik, *Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

APISAN: Sanitizing API Usages through Semantic Cross-checking

Insu Yun Changwoo Min Xujie Si Yeongjin Jang Taesoo Kim Mayur Naik
Georgia Institute of Technology

Abstract

API misuse is a well-known source of bugs. Some of them (e.g., incorrect use of SSL API, and integer overflow of memory allocation size) can cause serious security vulnerabilities (e.g., man-in-the-middle (MITM) attack, and privilege escalation). Moreover, modern APIs, which are large, complex, and fast evolving, are error-prone. However, existing techniques to help finding bugs require manual effort by developers (e.g., providing specification or model) or are not scalable to large real-world software comprising millions of lines of code.

In this paper, we present APISAN, a tool that automatically infers correct API usages from source code without manual effort. The key idea in APISAN is to extract likely correct usage patterns in four different aspects (e.g., causal relation, and semantic relation on arguments) by considering semantic constraints. APISAN is tailored to check various properties with security implications. We applied APISAN to 92 million lines of code, including Linux Kernel, and OpenSSL, found 76 previously unknown bugs, and provided patches for all the bugs.

1 Introduction

Today, large and complex software is built with many components integrated using APIs. While APIs encapsulate the internal state of components, they also expose rich semantic information, which renders them challenging to use correctly in practice. Misuse of APIs in turn leads to incorrect results and more critically, can have serious security implications. For example, a misuse of OpenSSL API can result in man-in-the-middle (MITM) attacks [22, 26], and seemingly benign incorrect error handling in Linux (e.g., missing a check on `kmalloc()`) can allow DoS or even privilege escalation attacks [12]. This problem, in fact, is not limited to API usage, but pervades the usage of all functions, which we generally refer to as APIs in this paper.

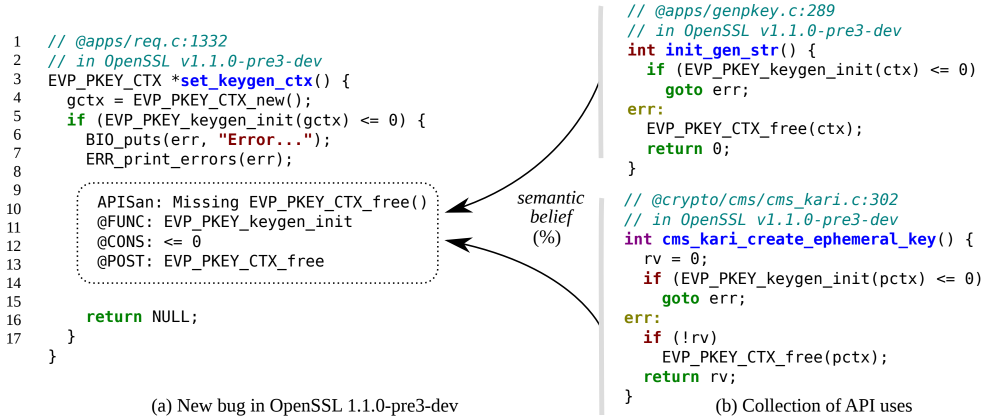
Many different tools, techniques, and methodologies have been proposed to address the problem of finding or preventing API usage errors. Broadly, all existing techniques either require (1) manual effort—API-specific specifications (e.g., SSL in SSLint [26], setuid [10, 15]), code annotations (e.g., lock operations in Sparse [41]), correct models (e.g., file system in WOODPECKER [11]), or (2) an accurate analysis of source code [6, 7], which is hard to scale to complex, real-world system software written in C/C++.

We present a fully automated system, called APISAN for finding API usage errors. Unlike traditional approaches that require API-specific specifications or models, APISAN infers the correct usage of an API from other uses of the API, regarding the majority usage pattern as a *semantic belief*, i.e., the likely correct use. Also, instead of relying on whole-program analysis, APISAN represents correct API usage in a probabilistic manner, which makes it scalable beyond tens of millions of lines of low-level system code like the Linux kernel. In APISAN, the higher the observed number of API uses, potentially even from different programs, the stronger is the belief in the inferred correct use. Once APISAN extracts such semantic beliefs, it reports deviations from the beliefs as potential errors together with a probabilistic ranking that reflects their likelihood.

A hallmark of APISAN compared to existing approaches [1, 18, 28, 29] for finding bugs by detecting contradictions in source code is that it achieves precision by considering semantic constraints in API usage patterns. APISAN infers such constraints in the form of symbolic contexts that it computes using a symbolic execution based technique. The technique, called *relaxed symbolic execution*, circumvents the path-explosion problem by limiting exploration to a bounded number of intra-procedural paths that suffice in practice for the purpose of inferring semantic beliefs.

APISAN computes a database of symbolic contexts from the source code of different programs, and infers semantic beliefs from the database by checking four key aspects: implications of function return values, relations between function arguments, causal relationships between functions, and implicit pre- and post-conditions of functions. These four aspects are specialized to incorporate API-specific knowledge for more precise ranking and deeper semantic analysis. We describe eight such cases in APISAN that are tailored to check a variety of properties with security implications, such as cryptographic protocol API misuses, integer overflow, improper locking, and NULL dereference.

Our evaluation shows that APISAN's approach is scal-



```
1   // @apps/req.c:1332
2   // in OpenSSL v1.1.0-pre3-dev
3   EVP_PKEY_CTX *set_keygen_ctx() {
4     gctx = EVP_PKEY_CTX_new();
5     if (EVP_PKEY_keygen_init(gctx) <= 0) {
6       BIO_puts(err, "Error...");
7       ERR_print_errors(err);
8
9
10       APISan: Missing EVP_PKEY_CTX_free()
11       @FUNC: EVP_PKEY_keygen_init
12       @CONS: <= 0
13       @POST: EVP_PKEY_CTX_free
14
15
16      return NULL;
17    }
    }
```

(a) New bug in OpenSSL 1.1.0-pre3-dev

```
// @apps/genpkey.c:289
// in OpenSSL v1.1.0-pre3-dev
int init_gen_str() {
  if (EVP_PKEY_keygen_init(ctx) <= 0)
    goto err;
err:
  EVP_PKEY_CTX_free(ctx);
  return 0;
}
```

```
// @crypto/cms/cms_kari.c:302
// in OpenSSL v1.1.0-pre3-dev
int cms_kari_create_ephemeral_key() {
  rv = 0;
  if (EVP_PKEY_keygen_init(pctx) <= 0)
    goto err;
err:
  if (!rv)
    EVP_PKEY_CTX_free(pctx);
  return rv;
}
```

(b) Collection of API uses

Figure 1: (a) A memory leak vulnerability found by APISan in OpenSSL 1.1.0-pre3-dev. When a crypto key fails to initialize, the allocated context (i.e., `gctx`) should be freed. Otherwise, a memory leak will occur. APISan first infers correct semantic usage of the API from (b) other uses of the API, and extracts a checkable rule, called a *semantic belief*, under the proper context (e.g., state: `EVP_PKEY_keygen_init()` → `rv <= 0` && `EVP_PKEY_CTX_free()`). This newly found vulnerability has been reported and fixed in the mainstream with the patch we provided. In the above report, `@FUNC` indicates a target API, `@CONS` is a return value constraint, and `@POST` shows an expected post-action following the API.

able and effective in finding API misuses that result in critical security problems such as code execution, system hangs, or crashes. In total, we analyzed 92 million lines of code (LoC) and found 76 *previously unknown* bugs in Linux, OpenSSL, PHP, Python, and debian packages using OpenSSL (see Table 2). More importantly, we created patches for all these bugs and sent them to the mainline developers of each project. Of these, 69 bugs have been confirmed, and most have already been applied to the mainstream repositories. We are awaiting responses for the remaining reported bugs.

In short, our paper makes the following contributions:

- **New methodology.** We develop a fully automated way of finding API misuses that infers semantic beliefs from existing API uses and probabilistically ranks deviant API usages as bugs. We also formalize our approach thoroughly.
- **Practical impact.** APISan found 76 new bugs in system software and libraries, including Linux, OpenSSL, PHP, and Python, which are 92 million LoC in total. We created patches for all bugs and most of them have already been fixed in the mainstream repositories of each project.
- **Open source tool.** We will make the APISan framework and all its checkers publicly available online for others to readily build custom checkers on top of APISan.

The rest of this paper is organized as follows. §2 provides an overview of APISan. §3 describes APISan's design. §4 presents various checkers of APISan. §5 describes APISan's implementation. §6 explains the bugs we found. §7 discusses APISan's limitations and potential future directions. §8 compares APISan to previous work and §9 concludes.

2 Overview

In this section, we present an overview of APISan, our system for finding API usage errors. These errors often have security implications, although APISan and the principles underlying it apply to general-purpose APIs and are not limited to finding security errors in them. To find API usage errors, APISan automatically infers semantic correctness, called *semantic beliefs*, by analyzing the source code of different uses of the API.

We motivate our approach by means of an example that illustrates an API usage error. We outline the challenges faced by existing techniques in finding the error and describe how APISan addresses those challenges.

2.1 Running Example

Figure 1(a) shows an example of misusing the API of OpenSSL. The allocated context of a public key algorithm (`gctx` on Line 3) must be initialized for a key generation operation (`EVP_PKEY_keygen_init()` on Line 4). If the initialization fails, the allocated context should be freed by calling `EVP_PKEY_CTX_free()`. Otherwise, it results in a memory leak.

To find such errors automatically, a checker has to know the correct usage of the API. Instead of manually encoding semantic correctness, APISan automatically infers the correct usage of an API from other uses of the API, regarding the majority usage pattern as the likely correct use. For example, considering the use of the OpenSSL API in Figure 1(a) together with other uses of the API shown in Figure 1(b), APISan infers the majority pattern as freeing the allocated context after initialization failure (i.e., `EVP_PKEY_keygen_init() <= 0`), and thereby reports the use in Figure 1(a) as an error.

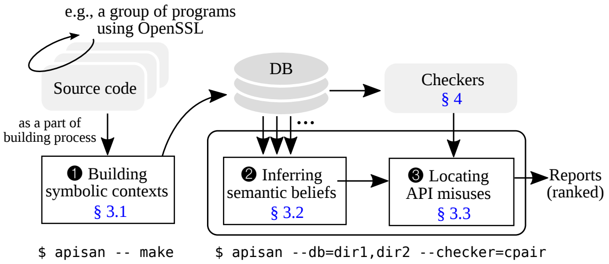
2.2 Challenges

We describe three key challenges that hinder existing approaches in finding the error in the above example.

1. Lack of specifications. A large body of work focuses on checking semantic correctness, notably dataflow analysis and model checking approaches [3, 4, 14, 17, 21, 46]. A major obstacle to these approaches is that developers should manually describe *"what is correct,"* and this effort is sometimes prohibitive in practice. To alleviate this burden, many of the above approaches check lightweight specifications, notably type-state properties [42]. These specifications are not expressive enough to capture correct API uses inferred by APISAN; for example, type-state specifications can capture finite-state rules but not rules involving a more complex state, such as the rule in the box in Figure 1(a), which states that `EVP_PKEY_CTX_free()` must be called if `EVP_PKEY_CTX_init() <= 0`. Moreover, techniques for checking such rules must track the context of the API use in order to be precise, which limits their scalability. For instance, the second example in Figure 1(b) has a constraint on `!rv`, whose tracking is necessary for precision but complicated by the presence of `goto` routines in the example.

2. Missing constraints. Engler et al. [18] find potential bugs by detecting contradictions in software in the absence of correctness semantics specified by developers. For instance, if most occurrences of a lock release operation are preceded by a lock acquire operation, then instances where the lock is released without being acquired are flagged as bugs. The premise of APISAN is similar in that the majority occurrence of an API usage pattern is regarded as likely the correct usage, and deviations are reported as bugs. However, Engler et al.'s approach does not consider semantic constraints, which can lead it to miss bugs that occur under subtle constraints, such as the one in Figure 1(a), which states that `EVP_PKEY_CTX_free()` must be called only when `EVP_PKEY_keygen_init()` fails.

3. Complex constraints. KLEE [7] symbolically executes all possible program paths to find bugs. While it is capable of tracking semantic constraints, however, it suffers from the notorious path-explosion problem; its successor, UC-KLEE [37], performs under-constrained symbolic execution that checks individual functions rather than whole programs. However, functions such as `EVP_PKEY_keygen_init()` in Figure 1 contain a function pointer, which is hard to resolve in static analysis, and cryptographic functions have extremely complex path constraints that pose scalability challenges to symbolic execution based approaches.


e.g., a group of programs using OpenSSL
Source code
as a part of building process
DB
Checkers
§ 4
❶ Building symbolic contexts
§ 3.1
❷ Inferring semantic beliefs
§ 3.2
❸ Locating API misuses
§ 3.3
Reports (ranked)

```
$ apisan -- make
$ apisan --db=dir1,dir2 --checker=cpair
```

Figure 2: Overview of APISAN's architecture and workflow. APISAN first builds symbolic contexts from existing programs' source code and creates a database (§3.1); then APISAN infers correct usages of APIs, so-called *semantic beliefs*, in four aspects (§3.2). The inferred beliefs are used to find and rank potential API misuses to be reported as bugs (§3.3). Specific checkers are built by using the inferred beliefs and symbolic context database. If necessary, checkers incorporate domain-specific knowledge to find and rank bugs more precisely (§4).

2.3 Our Approach

APISAN's workflow consists of three basic steps as shown in Figure 2. It first builds symbolic contexts using symbolic execution techniques on existing programs' source code and creates a database of symbolic traces (§3.1). Then, it statistically infers correct API usages, called semantic beliefs, using the database (§3.2). Finally, it locates API misuses in the programs' source code using the inferred beliefs and domain-specific knowledge if necessary (§3.3, §4).

We formalize our approach as a general framework, shown in Figure 5, which can be tuned using two parameters: the context checking function, which enables tailoring the checking of symbolic contexts to different API usage aspects, and an optional hint ranking function, which allows customizing the ranking of bug reports. As we will discuss shortly, our framework provides several built-in context checking functions, allowing common developers to use APISAN without modification.

Below, we describe how APISAN tackles the challenges outlined in the previous section.

1. Complete automation. In large and complex programs, it is prohibitive to rely on manual effort to check semantic correctness, such as manually provided specifications, models, or formal proofs. Instead, APISAN follows a fully automated approach, inferring semantic beliefs, i.e., correct API usages, from source code.

2. Building symbolic contexts. To precisely capture API usages involving a complex state, APISAN infers semantic beliefs from the results of symbolic execution. These results, represented in the form of symbolic constraints, on one hand contain precise semantic information about each individual use of an API, and on the other hand are abstract enough to compare across uses of the API even in different programs.

3. Relaxed symbolic execution. To prevent the path explosion problem and achieve scalability, we perform *relaxed symbolic execution*. Unlike previous approaches, which try to explore as many paths as possible, APISAN explores as few paths as possible so as to suffice for the purpose of inferring semantic beliefs. In particular, our relaxed symbolic execution does not perform interprocedural analysis, and unrolls loops.

4. Probabilistic ranking. To allow to prioritize developers' inspection effort, APISAN ranks more likely bug reports proportionately higher. More specifically, APISAN's ranking is probabilistic, denoting a confidence in each potential API misuse that is derived from a proportionate number of occurrences of the majority usage pattern, which itself is decided based on a large number of uses of the API in different programs. The ranking is easily extensible with domain-specific ranking policies for different API checkers.

3 Design of APISAN

The key insight behind our approach is that the "correctness" of API usages can be probabilistically measured from existing uses of APIs: that is, the more API patterns developers use in similar contexts, the more confidence we have about the correct API usage. APISAN automatically infers correct API usage patterns from existing source code without any human intervention (e.g., manual annotation or providing an API list), and ranks potential API misuses based on the extent to which they deviate from the observed usage pattern. To process complex, real-world software, APISAN's underlying mechanisms for inferring, comparing, and contrasting API usages should be scalable, yet without sacrificing accuracy. In this section, we elaborate on our static analysis techniques based on relaxed symbolic execution (§3.1), methodologies to infer semantically correct API usages (§3.2), and a probabilistic method for ranking potential API misuses (§3.3).

3.1 Building Symbolic Contexts

APISAN performs symbolic execution to build symbolic contexts that capture rich semantic information for each function call. The key challenge of building symbolic contexts in large and complex programs is to overcome the path-explosion problem in symbolic execution.

We made two important design decisions for our symbolic execution to achieve scalability yet extract accurate enough information about symbolic contexts. First, APISAN localizes symbolic execution within a function boundary. Second, APISAN unrolls each loop once so that the results of symbolic execution can be efficiently represented as a *symbolic execution tree* with no backward edges. In this section, we provide justifications for these two design decisions within the context of finding API misuses, and provide a performance optimization that memoizes the predominant symbolic states. Finally, we precisely define the structure of symbolic execution traces computed by APISAN.

```
  // @drivers/tty/synclink_gt.c:2363
  // in Linux v4.5-rc4
  static irqreturn_t slgt_interrupt(int dummy, void *dev_id) {
    struct slgt_info *d = dev_id;
    ...
    for (i = 0; i < d->count; i++) {
      if (d->ports[i] == NULL)
        continue;
*     spin_lock(&d->ports[i]->lock);
      ...
*     spin_unlock(&d->ports[i]->lock);
    }
    ...
    return IRQ_HANDLED;
  }
```

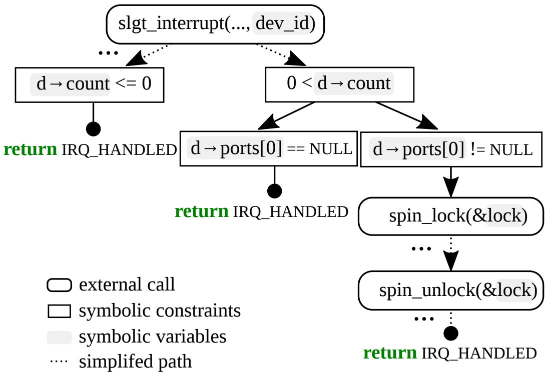


Figure 3: A typical API usage inside a loop. This code snippet comes from a `tty` device driver in the Linux v4.5-rc1. `spin_lock()` and `spin_unlock()` are used in a pair inside the loop. APISAN represents its symbolic context as a tree that contains function calls and symbolic constraints by unrolling its outer loop, as depicted at the bottom of the code snippet. Note that we use `lock` for `d->ports[0]->lock` due to space limitation.

Limiting inter-procedural analysis. In APISAN, we perform symbolic execution intra-procedurally for each function. We use a fresh symbolic variable to represent each formal argument of the function, as well as the return value of each function called in its body. The symbolic constraints track C/C++ expressions over such symbolic variables, as described below. In our experience with APISAN, limiting inter-procedural analysis is reasonable for accuracy and code coverage, since most API usages can be captured within a caller function without knowing API internals.

Unrolling a loop. APISAN unrolls each loop only once to reduce the number of paths explored. While this can limit the accuracy of our symbolic execution, it does not noticeably affect the accuracy of APISAN. This is because most API usages in practice do not tend to be related to loop variables. Figure 3 (top) shows such an example in a Linux device driver. Although the symbolic context changes while executing the loop, API usages

$$
\begin{array}{rcll}
\text{(function)} & f & \in & \mathbb{F} \\
\text{(integer)} & n & \in & \mathbb{Z}, \quad \text{(natural)} \quad i \in \mathbb{N} \\
\text{(symbolic variable)} & \alpha & ::= & \langle \textbf{arg}, i \rangle \mid \langle \textbf{ret}, i \rangle \\
\text{(symbolic expression)} & e & ::= & n \mid \alpha \mid uop\ e \mid e_1\ bop\ e_2 \\
\text{(integer range)} & r & ::= & [n_1, n_2] \\
\text{(event in trace)} & a & ::= & \textbf{call}\ f(\bar{e}) \mid \textbf{assume}(e, \bar{r}) \\
\text{(trace)} & t & ::= & \bar{a} \\
\text{(database of traces)} & \mathbb{D} & ::= & \{\, t_1, t_2, \cdots \} \\
\end{array}
$$

Figure 4: Abstract syntax of symbolic execution traces.

of `spin_lock()` and `spin_unlock()` can be precisely captured even by unrolling the loop once. While this may not always be the case, however, we compensate for the incurred accuracy loss by collecting a larger number of API uses.

Memoizing predominant symbolic states. Another advantage of loop unrolling is that all symbolic execution traces of a function can be efficiently represented as a tree, namely, a symbolic execution tree, without having backward edges. This helps scalability because APISAN can deterministically explore the symbolic execution tree, and all intermediate results can be cached in interior nodes; most importantly, the cached results (i.e., predominant symbolic contexts) can be safely re-used because there is no control flow from a child to its ancestors. Figure 3 (bottom) shows the corresponding symbolic execution tree for the function `slgt_interrupt` shown above it.

Structure of symbolic execution traces. Figure 4 formally describes the structure of traces computed by APISAN using symbolic execution. Each trace t consists of a sequence of events. We refer to the i^{th} event by $t[i]$, where $1 \leq i \leq |t|$. Each event a is either a **call** to a function f with a sequence of symbolic expressions $\bar{e}$ as arguments, or an **assume** constraint, which is a pair consisting of a symbolic expression e and its possible value ranges $\bar{r}$. A symbolic expression e can be a constant n, a symbolic variable α, or the result of an unary (uop) or binary (bop) operation on other symbolic expressions. Each symbolic variable α is either the return result of a function called at the i^{th} event in the trace, denoted $\langle \textbf{ret}, i \rangle$, or the i^{th} formal parameter of the function being symbolically executed, denoted $\langle \textbf{arg}, i \rangle$.

The following three traces are computed by APISAN for the code snippet in Figure 3 (ignoring unseen parts)[1]:

$$
\begin{array}{ll}
t_1: & \textbf{assume}(\mathsf{d{\rightarrow}count}, [\mathsf{MIN}, 0]) \\
t_2: & \textbf{assume}(\mathsf{d{\rightarrow}count}, [1, \mathsf{MAX}]); \\
 & \textbf{assume}(\mathsf{d{\rightarrow}ports[0]}, [0, 0]) \\
t_3: & \textbf{assume}(\mathsf{d{\rightarrow}count}, [1, \mathsf{MAX}]); \\
 & \textbf{assume}(\mathsf{d{\rightarrow}ports[0]}, [[\mathsf{MIN}, -1], [1, \mathsf{MAX}]]); \\
 & \textbf{call}\ \mathsf{spin\_lock(\&d{\rightarrow}ports[0]{\rightarrow}lock)}; \\
 & \textbf{call}\ \mathsf{spin\_unlock(\&d{\rightarrow}ports[0]{\rightarrow}lock)} \\
\end{array}
$$

3.2 Inferring Semantic Beliefs

The key challenge is to infer (most likely) correct API usages that are implicitly embedded in a large number of existing implementations. We call the inferred API usages "semantic beliefs," not only because they are believed to be correct by a dominant number of implementations, but also because they are used in semantically similar contexts (e.g., certain state or conditions). Therefore, the more frequent the API usage patterns we observe, the stronger is the semantic belief about the correctness of API usages. APISAN infers semantic beliefs by analyzing the surrounding symbolic contexts (§3.1) without developers' manual annotations or providing an API list.

In particular, APISAN focuses on exploring four common API context patterns.

- **Return value:** Not only does a function return the result of its computation, but it often implicates the status of the computation through the return value; for example, non-zero value in `glibc` and `PTR_ERR()` in the Linux kernel.
- **Argument:** There are semantic relations among arguments of an API; for example, the memory copy size should be smaller or equal to the buffer size.
- **Causality:** Two APIs can be causally related; for example, an acquired lock should be released at the end of critical section.
- **Conditions:** API semantics can imply certain pre- or post-conditions; for example, verifying a peer certificate is valid only if the peer certificate exists.

We give a formal description of these four patterns in Figure 6 and elaborate upon them in the rest of this section. Since APISAN infers semantic beliefs, which are probabilistic in nature, there could be false positives in bug reports. APISAN addresses this problem by providing a ranking scheme for developers to check the most probable bug reports first. Figure 5 formalizes this computation and §3.3 presents it in further detail.

3.2.1 Implication of Return Values

Return value is usually used to return the computation result (e.g. pointer to an object) or execution status (e.g., `errno`) of a function. Especially for system programming in C, certain values are conventionally used to represent execution status. In such cases, checking the return value (execution status) properly before proceeding is critical to avoid security flaws. For instance, if a program ignores checking the return value of memory allocation (e.g., `malloc()`), it might crash later due to NULL pointer dereference. In the OpenSSL library, since the result of establishing a secure connection is passed by a return value, programs that fail to check the return value properly are vulnerable to MITM attacks [22].

[1] MIN and MAX stand for the minimum and maximum possible values of a related type, respectively.

$$
\begin{array}{rcl}
\textsf{SymbolicContexts}\,(f) & = & \{\ (t,i,C)\ \mid\ t\in\mathbb{D}\ \wedge\ i\in[1..|t|]\ \wedge\ t[i]\equiv\mathbf{call}\,f(*)\ \wedge\ C=\textsc{Contexts}(t,i)\ \}\\
\textsf{Frequency}\,(f,c) & = & \{\ (t,i)\ \mid\ \exists C:\ c\in C\wedge(t,i,C)\in\textsf{SymbolicContexts}(f)\ \}\\
\textsf{Majority}\,(f) & = & \{\ c\ \mid\ |\textsf{Frequency}(f,c)|\ /\ |\textsf{SymbolicContexts}(f)|\geq\theta\ \}\\
\textsf{BugReports}\,(f) & = & \{\ (t,i,C)\ \mid\ (t,i,C)\in\textsf{SymbolicContexts}(f)\ \wedge\ C\ \cap\ \textsf{Majority}(f)=\emptyset\ \}\\
\textsf{BugReportScore}\,(f) & = & 1-|\textsf{BugReports}(f)|\ /\ |\textsf{SymbolicContexts}(f)|\ +\ \textsc{Hint}(f)
\end{array}
$$

Figure 5: The general framework of APISAN. Threshold ratio θ is used to decide whether a context c is a correct or buggy API usage. Procedures CONTEXTS and HINT are abstract; Figure 6 shows concrete instances of these procedures implemented in APISAN.

$$
\begin{array}{rcl}
\textsf{returnValueContexts} & = & \lambda(t,i).\ \{\ \bar{r}\mid\exists j:\ t[j]\equiv\mathbf{assume}(e,\bar{r})\ \wedge\ \langle\mathbf{ret},i\rangle\in \mathit{retvars}(e)\ \}\\
\textsf{argRelationContexts} & = & \lambda(t,i).\ \{\ (u,v)\mid t[i]\equiv\mathbf{call}*(\bar{e})\ \wedge\ \mathit{argvars}(\bar{e}[u],t)\ \cap\ \mathit{argvars}(\bar{e}[v],t)\neq\emptyset\ \}\\
\textsf{causalityContexts}\langle\bar{r}\rangle & = & \lambda(t,i).\ \{\ g\mid\exists j:t[j]\equiv\mathbf{assume}(e,\bar{r})\ \wedge\ \langle\mathbf{ret},i\rangle\in\mathit{retvars}(e)\ \wedge\ \exists k>j:t[k]\equiv\mathbf{call}\,g(*)\ \}\\
\textsf{conditionContexts}\langle\bar{r}\rangle & = & \lambda(t,i).\ \{\ (g,\bar{r}')\mid\exists j:t[j]\equiv\mathbf{assume}(e,\bar{r})\ \wedge\ \langle\mathbf{ret},i\rangle\in\mathit{retvars}(e)\ \wedge\ \exists k>j:t[k]\equiv\mathbf{call}\,g(*)\ \wedge\\
 & & \qquad\exists l:t[l]\equiv\mathbf{assume}(e',\bar{r}')\wedge\langle\mathbf{ret},k\rangle\in\mathit{retvars}(e')\ \}\\
\textsf{defaultHint} & = & \lambda f.\ 0 \qquad \textsf{nullDerefHint}\ =\ \lambda f.\ \text{if }(f\text{'s name contains }\mathit{alloc})\text{ then }0.3\text{ else }0
\end{array}
$$

Figure 6: Concrete instances of the CONTEXTS and HINT procedures implemented in APISAN. Function *retvars*(e) returns all $\langle\mathbf{ret},i\rangle$ variables in e. Function *argvars*(e,t) returns all $\langle\mathbf{arg},i\rangle$ variables in e, consulting t to recursively replace each $\langle\mathbf{ret},i\rangle$ variable by its associated function call symbolic expression. Both these functions are formally described in Appendix A.

Moreover, missing return value checks can lead to privilege escalation like `CVE-2014-4113` [12]. Because of such critical scenarios, `gcc` provides a special pragma, `__attribute__((warn_unused_result))`, to enforce the checking of return values. However, it does not guarantee if a return value check is proper or not [24].

Properly checking return values seems trivial at the outset, but it is not in reality; since each API uses return values differently (e.g., `0` can be used to denote either success or failure), it is error-prone. Figure 7 shows such an example found by APISAN in Linux. In this case, `kthread_run()` returns a new `task_struct` or a non-zero error code, so the check against `0` is incorrect (Line 12).

Instead of analyzing API internals, APISAN analyzes how return values are checked in different contexts to infer proper checking of return values of an API. For an API function `f`, APISAN extracts all symbolic constraints on `f`'s return values from symbolic execution traces. After extracting all such constraints, APISAN calculates the probability of correct usage for each constraint based on occurrence count. For example, APISAN extracts how frequently the return value of `kthread_run()` is compared with `0` or `IS_ERR(p)`. APISAN reports such cases that the probability of constraints is below a certain threshold as potential bugs; the lower the probability of correctness, the more likely those cases are to be bugs.

Our framework can be easily instantiated to capture return value context by defining the context function $\textsf{returnValueContexts}(t,i)$, as shown in Figure 6, which extracts all checks on the return value of the function called at $t[i]$ (i.e., the i^{th} event in trace t).

3.2.2 Relations on Arguments

In many APIs, arguments are semantically inter-related. Typical examples are memory copy APIs, such as `strncpy(d,s,n)` and `memcpy(d,s,n)`; for correct operation without buffer overrun, the size of the destination buffer `d` should be larger or equal to the copy length `n`.

APISAN uses a simple heuristic to capture possible relations between arguments. APISAN decides that two arguments are related at a function call if their symbolic expressions share a common symbolic variable. For example, the first and third arguments of `strncpy(malloc(n+1),s,n)` are considered to be related. After deciding whether a pair of arguments are related or not at each call to a function, APISAN calculates the probability of the pair of arguments being related. APISAN then classifies the calls where the probability is lower than a certain threshold as potential bugs.

Another important type of relation on arguments is the constraint on a single argument, e.g., an argument is expected to be a format string. When such constraints exist on well-known APIs like `printf()`, they can be checked by compilers. However, a compiler cannot check user-defined functions that expect a format string argument.

To capture relations on arguments, we define the context function $\textsf{argRelationContexts}$ as shown in Figure 6. It is also straightforward to handle the format string check by extending the definition with a format check as a pair relation, such as $(-1,i)$, where -1 indicates that the pair is a special check and i denotes the i^{th} argument that is under consideration for a format check.

3.2.3 Constrained Causal Relationships

Causal relationships, also known as the `a-b` pattern, are common in API usage, such as `lock/unlock` and `malloc/free`. Past research [18, 29] only focuses on finding "direct" causal relationships, that is, no context constraint between two API calls. In practice, however, there are many *constrained* causal relationships as well. The conditional synchronization primitives shown in Figure 8 are one such example. In this case, there is a causal relationship between `mutex_trylock()` and `mutex_unlock()` only when `mutex_trylock()` returns a non-zero value.

```
// @drivers/media/usb/pvrusb2/pvrusb2-context.c:194
// in Linux v4.5-rc4
int pvr2_context_global_init(void) {
  pvr2_context_thread_ptr = \
         kthread_run(pvr2_context_thread_func,
         NULL,
         "pvrusb2-context");
  // APISan: Incorrect return value check
  // @FUNC: kthread_run
  // @CONS: >= (unsigned long)-4095
  //        <  (unsigned long)-4095
★ return (pvr2_context_thread_ptr ? 0 : -ENOMEM);
}
```

Figure 7: Incorrect handling of a return value in Linux found by APISAN. `kthread_run()` returns a pointer to `task_struct` upon success or returns an error code upon failure. Because of incorrect handling of return values, this function always returns `0`, i.e., success, even in the case of error.

Both direct and constrained causality relationships can be effectively captured in the APISAN framework by defining a parametric context function causalityContexts$\langle \bar{r} \rangle$ shown in Figure 6, which extracts all pairs of API calls with $\bar{r}$ as the context constraints between them. Conceptually, the parameter $\bar{r}$ is obtained by enumerating all constraints on return values from all symbolic execution traces. In practice, however, we only check $\bar{r}$ when necessary, for example, we only check constraints on the return value of `f()` after a call to `f()`.

3.2.4 Implicit Pre- and Post-Conditions

In many cases, there are hidden assumptions *before* or *after* calling APIs, namely, implicit pre- and post-conditions. For example, the memory allocation APIs assume that there is no integer overflow on the argument passed as allocation size, which implies that there should be a proper check before the call. Similarly, `SSL_get_verify_result()`, an OpenSSL API which verifies the certificate presented by the peer, is meaningful only when `SSL_get_peer_certificate()` returns a non-NULL certificate of a peer, though which could happen either before or after `SSL_get_verify_result()`. So the validity check of a peer certificate returned by `SSL_get_peer_certificate()` is an implicit pre- or post-condition of `SSL_get_verify_result()`.

Similar to the context checking of causal relationships, we define a parametric context function conditionContexts$\langle \bar{r} \rangle$ shown in Figure 6, to capture implicit pre- and post-conditions of an API call. Here, the parameter $\bar{r}$ serves as the pre-condition, and the post-condition is extracted along with the called API.

3.3 Ranking Semantic Disbeliefs

After collecting the API usage patterns discussed above, APISAN statistically infers the majority usage patterns for each API function under each context. This computation is described in detail in Figure 5. Intuitively, APISAN labels an API usage pattern as majority (i.e., likely correct usage) if its occurrence ratio is larger than a threshold θ. In our experience, this simple approach is quite effective, though more sophisticated statistical approaches could be further applied. Each call to a function that deviates from its majority usage pattern is reported as a potential bug.

```
// @kernel/workqueue.c:1977
// in Linux v4.5-rc4
static bool manage_workers(struct worker *worker)
{
    struct worker_pool *pool = worker->pool;
    if (!mutex_trylock(&pool->manager_arb))
        return false;
    pool->manager = worker;
    maybe_create_worker(pool);
    pool->manager = NULL;
    mutex_unlock(&pool->manager_arb);
    return true;
}
```

Figure 8: An example usage of conditional locking in Linux. `mutex_trylock()` returns non-zero value when a lock is acquired. So `mutex_unlock()` is necessary only in this case.

Since our approach is probabilistic in nature, a bug report found by APISAN might be a false alarm. APISAN ranks bug reports in decreasing order of their likelihood of being bugs, so that the most likely bugs have the highest priority to be investigated. Based on the observation that the more the majority patterns repeat, the more confident we are that these majority patterns are correct specifications, APISAN uses the ratio of majority patterns over "buggy" patterns as a measure of the likelihood. In addition, APISAN can also adjust the ranking with domain-specific knowledge about APIs. For example, if an API name contains a sub-string `alloc`, which indicates that it is very likely to handle memory allocation, we can customize APISAN to give more weight for its misuse in the return value checking.

4 Checking API Misuses

In this section, we demonstrate how inferred semantic beliefs described in the previous section can be used to find API misuses. In particular, we introduce eight cases, which use API-specific knowledge for more precise ranking and deeper semantic analysis.

4.1 Checking SSL/TLS APIs

A recent study shows that SSL/TLS APIs are very error-prone—especially, validating SSL certificates is *"the most dangerous code in the world"* [22]. To detect their incorrect use, specialized checkers that rely on hand-coded semantic correctness have been proposed [22, 26].

In APISAN, we easily created a SSL/TLS checker based on the constraints of return values and implicit pre- and post-conditions without manually coding seman-

```
  // @librabbitmq/amqp_openssl.c:180
  // in librabbitmq v0.8
  static int
  amqp_ssl_socket_open(void *base, const char *host,
                       int port, struct timeval *timeout) {
      // APISan: Missing implicit condition
      // @FUNC : SSL_get_verify_result
      // @CONS : == X509_V_OK
      // @COND : SSL_get_peer_certificate != NULL
+     cert = SSL_get_peer_certificate(self->ssl);
      result = SSL_get_verify_result(self->ssl);
-     if (X509_V_OK != result) {
+     if (!cert || X509_V_OK != result) {
        goto error_out3;
      }
  }
```

Figure 9: Incorrect use of OpenSSL API found in librabbitmq, a message queuing protocol library, by APISAN. `SSL_get_verify_result()` always returns `X509_V_OK` if there is no certificate (i.e., `!cert`). So `SSL_get_peer_certificate()` needs to be validated before or after calling `SSL_get_verify_result()`.

```
// @fs/ext4/resize.c:193
// in Linux v4.5-rc4
static struct ext4_new_flex_group_data
    *alloc_flex_gd(unsigned long flexbg_size)
{
    if (flexbg_size >=
        UINT_MAX / sizeof(struct ext4_new_flex_group_data))
        goto out2;
    flex_gd->count = flexbg_size;
    // APISan: Incorrect integer overflow check
    // @CONS: flexbg_size < UINT_MAX / 20
    // @EXPR: flexbg_size * 40
    flex_gd->groups =
        kmalloc(sizeof(struct ext4_new_group_data) *
                flexbg_size, GFP_NOFS);
}
```

Figure 10: An integer overflow vulnerability found in Linux by APISAN. Since `struct ext4_new_group_data` is larger than `struct ext4_new_flex_group_data`, previous overflow check can be bypassed. Interestingly, this bug was previously found by KINT and already patched [8], but APISAN found the patch is actually incorrect.

tic correctness. In practice, as we described in §3.2.4, the sequence of API calls and relevant constraints to validate SSL certificates can be captured by using implicit pre- and post-conditions. For example, Figure 9 shows that APISAN successfully inferred valid usage of `SSL_get_verify_result()` and discovered a bug.

4.2 Checking Integer Overflow

Integer overflows remain a very important threat despite extensive research efforts for checking them. Checkers have to deal with two problems: (1) whether there is a potential integer overflow, and (2) whether such a potential integer overflow is exploitable. KINT [45], the state-of-the-art integer security checker, relies on scalable static analysis to find potential integer overflows. To decide exploitability, KINT relies on users' annotations on untrusted data source and performs taint analysis to decide whether untrusted sources are related to an integer overflow. But if annotations are missing, KINT may miss some bugs.

Instead of annotating untrusted sources, APISAN infers *untrusted sinks* to decide that an integer overflow has security implications. The background belief is "checking sinks implies that such sinks are untrusted." APISAN considers APIs with arguments that are untrusted sinks as *integer overflow-sensitive APIs*. To infer whether an API is integer overflow-sensitive, the checker extracts all function calls whose arguments have arithmetic operations that can result in integer overflow. The checker classifies such function calls into three categories: (1) correct check, (2) incorrect check, and (3) missing check. If an argument has a constraint that prevents integer overflow, then it is a correct check. Determining potential integer overflow is straightforward because APISAN maintains a numerical range for each symbolic variable. If such a constraint cannot prevent integer overflow, then it is an incorrect check. Finally, if there is no constraint, then it is a missing check. The checker concludes that an API is more integer overflow-sensitive if the ratio of correct checks over total checks is higher. The checker gives a higher rank to incorrect checks followed by missing checks. For example, Figure 10 shows an integer overflow vulnerability found by APISAN.

4.3 Checking Memory Leak

A memory leak can be represented as a causal relationship between memory allocation and free functions. As Figure 1 shows, APISAN can infer a constrained causal relation between such a pair of functions, which may not be captured as a direct causal relation. When a function that is presumed to be a free function is not called following a function that is presumed to be the corresponding allocation function, it is reported as a memory leak with a higher rank. In this manner, APISAN effectively captures typical usage patterns of memory allocation and free routines to report potential memory leaks.

4.4 Checking Lock and Unlock

Similar to checking memory leaks, lock checking is based on a constrained causal relationship between lock and unlock functions inferred by APISAN. It gives a higher rank to cases where there are missing unlock function calls in some of the paths. For example, Figure 11 shows that there is one missing `clk_prepare_unlock()` call among two symbolic execution paths.

4.5 Checking NULL Dereference

NULL dereference can happen by accessing a pointer returned by a memory allocation function, such as `malloc()` and `kmalloc()`, without validation. Checking NULL

```
// @drivers/clk/clk.c:2672
// in Linux v4.5-rc4
void clk_unregister(struct clk *clk) {
  clk_prepare_lock();
  if (clk->core->ops == &clk_nodrv_ops) {
    pr_err("%s: unregistered clock: %s\n", __func__,
        clk->core->name);
    // APISan: Missing clk_prepare_unlock()
    // @FUNC: clk_prepare_lock
    // @CONS: None
    // @POST: clk_prepare_unlock
    return;
  }
  clk_prepare_unlock();
}
```

Figure 11: A missing unlock bug in Linux found by APISAN. It shows a common pattern of violating a causal relation.

dereference is based on the return value inference of APISAN. It collects how frequently the return value of a function is compared against NULL. Based on this information, it can find missing NULL checks. In addition, it gives a higher rank to cases where the function name contains common keywords for allocation such as `alloc` or `new`.

4.6 Checking Return Value Validation

Checking a return value of a function properly is more important than checking a return value itself. If the return value is incorrectly checked, the caller is likely to believe that the callee succeeded. Moreover, it is quite usual that incorrect checks fail only in rare cases, so that finding such incorrect checks is much more difficult than completely omitted checks. APISAN can find bugs of this kind, such as the one shown in Figure 7, by comparing constraints of return value checks.

4.7 Checking Broken Argument Relation

We can find potential bugs by inferring and finding broken relations between arguments. However, detecting a broken relation does not mean that it is always a bug, because there might be an implicit relation between two arguments that cannot be captured by APISAN (e.g., complex pointer aliasing of the buffer). This lack of information is complemented by a ranking policy that incorporates domain-specific knowledge, for example, a broken argument relation is ranked higher if either argument has a `sizeof()` operator.

4.8 Checking Format String

Incorrect use of format strings is one frequent source of security vulnerabilities [39]. Modern compilers (e.g., `gcc`) give compile-time warnings for well-known APIs such as `printf()`. However, in the case of programs that have their own `printf`-like functions (e.g., PHP), compilers cannot detect such errors.

To infer whether a function argument is a format string, we use a simple heuristic: if the majority of symbolic expressions for an argument is a constant string and contains well-known format codes (e.g, `%s`), then the argument is considered as a format string. For the cases where a symbolic variable is used as a format string argument, the corresponding API calls will be considered as potential bugs. Similarly, domain-specific knowledge can be applied as well. Bug reports of an API whose name contains a sub-string `print` is ranked higher, since it indicates that the API is very likely to take a format string as an argument.

5 Implementation

APISAN is implemented in 9K lines of code (LoC) as shown in Table 1: 6K of C/C++ for generating symbolic execution traces, which is based on `Clang` 3.6, and 3K of Python for checkers and libraries. We empirically chose a threshold value of **0.8** for deciding whether to label an API usage pattern as majority. Since APISAN ranks all reports in order of bug likelihood, however, the result is not sensitive to the threshold value in that the ordering of the top-ranked reports remains the same.

Component	Lines of code
Symbolic database generator	6,256 lines of C/C++
APISAN Library	1,677 lines of Python
Checkers	1,047 lines of Python
Total	8,980 lines of code

Table 1: Components and lines of code of APISAN.

6 Evaluation

To evaluate APISAN, this section attempts to answer the following questions:

- How effective is APISAN in finding previously unknown API misuses? (§6.1)
- How easy is APISAN to use by end-users and checker developers? (§6.2)
- How reasonable is APISAN's relaxed symbolic execution in finding bugs? (§6.3)
- How effective is APISAN's approach in ranking bugs? (§6.4)
- How effective is APISAN's approach compared to manual checking? (§6.5)

6.1 New Bugs

We applied APISAN to Linux v4.5-rc4, OpenSSL 1.1.0-pre3-dev, PHP 7.0, Python 3.6, and all 1,204 debian packages using the OpenSSL library. APISAN generated 40,006 reports in total, and we analyzed the reports

Program	Module	API misuse	Impact	Checker	#bugs	S.
Linux	cifs/cifs_dfs_ref.c	heap overflow	code execution	`args`	1	✓
	xenbus/xenbus_dev_frontend.c	missing integer overflow check	code execution	`intovfl`	1	✓
	ext4/resize.c	incorrect integer overflow check	code execution	`intovfl`	1	✓
	tipc/link.c	missing `tipc_bcast_unlock()`	deadlock	`cpair`	1	✓
	clk/clk.c	missing `clk_prepare_unlock()`	deadlock	`cpair`	1	✓
	hotplug/acpiphp_glue.c	missing `pci_unlock_rescan_remove()`	deadlock	`cpair`	1	✓
	usbvision/usbvision-video.c	missing `mutex_unlock()`	deadlock	`cpair`	1	✓
	drm/drm_dp_mst_topology.c	missing `drm_dp_put_port()`	DoS	`cpair`	1	✓
	affs/file.c	missing `kunmap()`	DoS	`cpair`	1	✓
	acpi/sysfs.c	missing `kobject_create_and_add()` check	system crash	`rvchk`	1	✓
	cx231xx/cx231xx-417.c	missing `kmalloc()` check	system crash	`rvchk`	1	✓
	qxl/qxl_kms.c	missing `kmalloc()` check	system crash	`rvchk`	1	P
	chips/cfi_cmdset_0001.c	missing `kmalloc()` check	system crash	`rvchk`	1	✓
	ata/sata_sx4.c	missing `kzalloc()` check	system crash	`rvchk`	1	✓
	hsi/hsi.c	missing `kzalloc()` check	system crash	`rvchk`	2	✓
	mwifiex/sdio.c	missing `kzalloc()` check	system crash	`rvchk`	2	✓
	usbtv/usbtv-video.c	missing `kzalloc()` check	system crash	`rvchk`	1	✓
	cxgb4/clip_tbl.c	missing `t4_alloc_mem()` check	system crash	`rvchk`	1	✓
	devfreq/devfreq.c	missing `devm_kzalloc()` check	system crash	`rvchk`	2	✓
	i915/intel_dsi_panel_vbt.c	missing `devm_kzalloc()` check	system crash	`rvchk`	1	✓
	gpio/gpio-mcp23s08.c	missing `devm_kzalloc()` check	system crash	`rvchk`	1	✓
	drm/drm_crtc.c	missing `drm_property_create_range()` check	system crash	`rvchk`	13	✓
	gma500/framebuffer.c	missing `drm_property_create_range()` check	system crash	`rvchk`	1	✓
	emu10k1/emu10k1_main.c	missing `kthread_create()` check	system crash	`rvchk`	1	✓
	m5602/m5602_s5k83a.c	missing `kthread_create()` check	system crash	`rvchk`	1	✓
	hisax/isdnl2.c	missing `skb_clone()` check	system crash	`rvchk`	1	✓
	qlcnic/qlcnic_ctx.c	missing `qlcnic_alloc_mbx_args()` check	system crash	`rvchk`	1	✓
	xen-netback/xenbus.c	missing `vzalloc()` check	system crash	`rvchk`	1	✓
	i2c/ch7006_drv.c	missing `drm_property_create_range()` check	system crash	`rvchk`	1	✓
	fmc/fmc-fakedev.c	missing `kmemdup()` check	system crash	`rvchk`	1	P
	rc/igorplugusb.c	missing `rc_allocate_device()` check	system crash	`rvchk`	1	✓
	s5p-mfc/s5p_mfc.c	missing `create_singlethread_workqueue()` check	system crash	`rvchk`	1	P
	fusion/mptbase.c	missing `create_singlethread_workqueue()` check	system crash	`rvchk`	1	P
	nes/nes_cm.c	missing `create_singlethread_workqueue()` check	system crash	`rvchk`	1	✓
	dvb-usb-v2/mxl111sf.c	missing `mxl111sf_enable_usb_output()` check	malfunction	`rvchk`	2	✓
	misc/xen-kbdfront.c	missing `xenbus_printf()` check	malfunction	`rvchk`	1	✓
	pvrusb2/pvrusb2-context.c	incorrect `kthread_run()` check	malfunction	`rvchk`	1	P
	agere/et131x.c	incorrect `drm_alloc_coherent()` check	malfunction	`rvchk`	1	✓
	drbd/drbd_receiver.c	incorrect `crypto_alloc_hash()` check	malfunction	`rvchk`	1	✓
	mlx4/mr.c	incorrect `mlx4_alloc_cmd_mailbox()` check	maintanence	`rvchk`	1	✓
	usnic/usnic_ib_qp_grp.c	incorrect `kzalloc()` check	maintanence	`rvchk`	2	✓
	aoe/aoecmd.c	incorrect `kthread_run()` check	maintanence	`rvchk`	1	✓
	ipv4/tcp.c	incorrect `crypto_alloc_hash()` check	maintanence	`rvchk`	1	✓
	mfd/bcm590xx.c	incorrect `i2c_new_dummy()` check	maintanence	`rvchk`	1	P
	usnic/usnic_ib_main.c	incorrect `ib_alloc_device()` check	maintanence	`rvchk`	1	✓
	usnic/usnic_ib_qp_grp.c	incorrect `usnic_fwd_dev_alloc()` check	maintanence	`rvchk`	1	✓
OpenSSL	dsa/dsa_gen.c	missing `BN_CTX_end()`	DoS	`cpair`	1	✓
	apps/req.c	missing `EVP_PKEY_CTX_free()`	DoS	`cpair`	1	✓
	dh/dh_pmeth.c	missing `OPENSSL_memdup()` check	system crash	`rvchk`	1	✓
PHP	standard/string.c	missing integer overflow check	code execution	`intovfl`	3	✓
	phpdbg/phpdbg_prompt.c	format string bug	code execution	`args`	1	✓
Python	Modules/zipimport.c	missing integer overflow check	code execution	`intovfl`	1	✓
rabbitmq	librabbitmq/amqp_openssl.c	incorrect `SSL_get_verify_result()` use	MITM	`cond`	1	✓
hexchat	common/server.c	incorrect `SSL_get_verify_result()` use	MITM	`cond`	1	✓
lprng	auth/ssl_auth.c	incorrect `SSL_get_verify_result()` use	MITM	`cond`	1	P
afflib	lib/aftest.cpp	missing `BIO_new_file()` check	system crash	`rvchk`	1	✓
	tools/aff_bom.cpp	missing `BIO_new_file()` check	system crash	`rvchk`	1	✓

Table 2: List of new bugs discovered by APISAN. We sent patches of all 76 new bugs; 69 bugs have been already confirmed and applied by corresponding developers (marked ✓ in the rightmost column); 7 bugs (marked P in the rightmost column) have not been confirmed yet. APISAN analyzed 92 million LoC and found one bug per 1.2 million LoC.

according to ranks. As a result, APISAN found 76 previously unknown bugs: 64 in Linux, 3 in OpenSSL, 4 in PHP, 1 in Python, and 5 in the debian packages (see Table 2 for details). We created patches for all the bugs and sent them to the mainline developers of each project. 69 bugs have been confirmed by the developers and most have already been applied to the mainline repositories. For remaining, 7 bugs, we are waiting for their response.

Security implications. All of the bugs we found have serious security implications: e.g., code execution, system crash, MITM, etc. For a few bugs including integer overflows in Python(`CVE-2016-5636` [13]) and PHP, we could even successfully exploit them by chaining ROP gadgets [2, 27]. In addition, we found that the vulnerable Python module is in the whitelist of Google App Engine and reported it to Google.

6.2 Usability

End-users. APISAN can be seamlessly integrated into an existing build process. Users can generate symbolic execution databases by simply invoking the existing build command, e.g., `make`, with `apisan`.

```
# generate DB
$ apisan make
```

With the database, users can run various checkers, which extract semantic beliefs from the database and locate potential bugs in order of their likelihood. For eight types of API misuses described at §4, we developed five checkers: return value checker (`rvchk`), causality checker (`cpair`), argument relation checker (`args`) implicit pre- and post-condition checker (`cond`), and integer overflow checker (`intovfl`).

```
# run a causality checker
$ apisan --checker=cpair
@FUNC: EVP_PKEY_keygen_init
@CONS: ((-2147483648, 0),)
@POST: EVP_PKEY_CTX_free
@CODE: {'req.c:1745'}
...
```

APISAN can also be run against multiple databases generated by different project code repositories. For example, users can infer semantic beliefs from multiple programs (e.g., all packages using `libssl`) and similarly get a list of ranked, potential bugs. This is especially useful for relatively young projects, which lack sufficient API usages.

```
# check libssl misuses by using rabbitmq and hexchat repos
$ apisan --checker=cond --db=rabbitmq,hexchat
```

Checker developers. Developing specialized checkers is easy; APISAN provides a simple interface to access symbolic execution databases. Each of our checkers is around 200 lines of Python code as shown in §5. Providing API-specific knowledge such as manual annotations can be easily integrated in the Python script.

		UC-KLEE	APISAN
Approach	Loop	best effort	once
	Inter-procedural	yes	no
	Constraint	SAT	numerical range
Bugs (OpenSSL)	Memory leak	5	7 (2*)
	NULL dereference	-	11
	Uninitialized data	6	-

Table 3: Comparison between UC-KLEE and APISAN in approaches and bugs found in OpenSSL v1.0.2, which is used in UC-KLEE's evaluation [37]. APISAN found 7 memory leak bugs and 11 NULL dereference vulnerabilities; two memory leak bugs (marked ⋆) were previously unknown, and our two patches have been applied to the mainline repository.

6.3 Effect of Relaxed Symbolic Execution

One of our key design decisions is to use relaxed symbolic execution for scalability at the cost of accuracy. To evaluate the effect of this design decision, we compare APISAN against UC-KLEE, which performs best-effort accurate symbolic execution including inter-procedural analysis and best-effort loop unrolling. For comparison, we ran UC-KLEE and APISAN on OpenSSL v1.0.2, which is the version used for UC-KLEE's evaluation. Table 3 shows a summary of the result.

APISAN found 11 NULL dereference bugs caused by missing return value checks of `OPENSSL_malloc()`, which are already fixed in the latest OpenSSL. Also, APISAN found seven memory leak bugs related to various APIs, such as `BN_CTX_new()`, `BN_CTX_start()`, and `EVP_PKEY_CTX_new()`, without any annotations. Two of these bugs were previously unknown; we sent patches which were confirmed and applied to the OpenSSL mainline. UC-KLEE found five memory leak bugs related to `OPENSSL_malloc()` with the help of users' annotations.

Interestingly, there is no common bug between UC-KLEE and APISAN. UC-KLEE cannot find the bugs that APISAN found because of function pointers, which are frequently used for polymorphism, and path explosion in complex cryptographic operations. APISAN does not discover the five memory bugs that UC-KLEE found because of diverse usages of `OpenSSL_malloc()`. Also, APISAN could not find any uninitialized memory bugs since it does not track memory accesses.

6.4 Ranking Effectiveness

Another key design aspect of APISAN is its ranking scheme. In this section, we investigate two aspects of our ranking scheme: (1) where true-positives are located in bug reports and (2) what are typical reasons of false positives. To this end, we analyzed the results of the return value checker (`rvchk`) on Linux v4.5-rc4.

True positives. If true-positive reports are highly ranked, developers can save effort in investigating bug reports. An

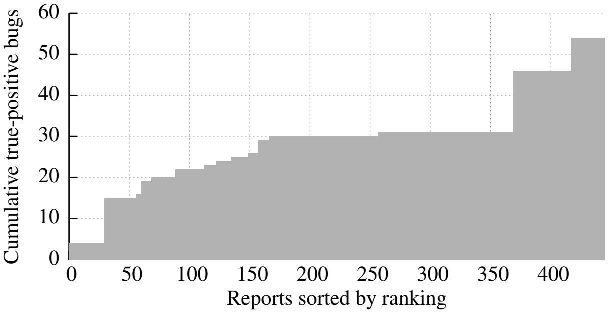


Figure 12: Cumulative true-positive bugs in Linux v4.5-rc4 reported by our return value checker (`rvchk`). We investigated top 445 bug reports out of 2,876 reports in total. Most new bugs are highly ranked.

author audited the top 445 reports out of 2,876 reports for two days and found 54 new bugs. As shown in Figure 12, most new bugs are highly ranked. This shows that our ranking scheme is effective to save developers' effort by letting them investigate the highest-ranked reports first.

False positives. To understand what causes false positives, we manually investigated all false positive cases in the top 445 reports, and found a few frequent reasons: diverse patterns of return value checking, wrapper functions delegating return value checking to callers, and semantically correct, but rare patterns.

Some kernel APIs, such as `snd_pcm_new()` [40], return zero on success or a negative error code on failure. In this case, there are two valid ways to check for error: comparison against zero (i.e., == `0`) or negative value (i.e., < `0`). If the majority of code follows one pattern (`snd_pcm_new()` <`0`), APISAN flags the minor correct cases as bugs.

Some wrapper functions delegate return value checking to their callers. APISAN treats these cases as if return value checking is missing because APISAN does not perform inter-procedural analysis.

If a return value of a function can have multiple meanings, APISAN can decide the rare cases as bugs. For example, most functions use `strcmp()` to test if two strings are equivalent (i.e., == `0`). But for the rare cases, which in fact use `strcmp()` to decide alphabetical order of two strings (i.e., < `0`), APISAN generates false alarms.

6.5 Comparison with Manual Auditing

The other extreme to automatic bug finding is manual auditing by developers. Manual auditing would be the most accurate but is not scalable in size and cost. We compared APISAN with manual auditing to grasp how accurate APISAN is compared to the ground truth.

To this end, we manually inspected memory allocation and free functions in OpenSSL v1.1.0-pre3-dev because OpenSSL faithfully follows naming conventions: allocation functions end with `_new` or `alloc`, and free functions end with `_free`.

```
  // @ext/standard/string.c:877
  // in PHP v5.5.9-rc1
  PHP_FUNCTION(wordwrap) {
    if (linelength > 0) {
      chk = (int)(textlen/linelength + 1);
      // no integer overflow
      newtext = safe_emalloc(chk, \
          breakcharlen, textlen + 1);
      alloced = textlen + chk * breakcharlen + 1;
    }
  }
```

```
  // @ext/standard/string.c:946
  // in PHP v7.0.0-rc1
  PHP_FUNCTION(wordwrap) {
    if (linelength > 0) {
      chk = (size_t)(ZSTR_LEN(text)/linelength + 1);
      // introduce a new integer overflow
*     newtext = zend_string_alloc( \
*         chk * breakchar_len + ZSTR_LEN(text), 0);
      alloced = ZSTR_LEN(text) + chk * breakchar_len + 1;
    }
  }
```

Figure 13: An integer overflow bug introduced by changing string allocation API in PHP. While the old string allocation API, `safe_emalloc()`, internally checks integer overflow, the new API, `zend_string_alloc()` has no such check.

To detemine how APISAN accurately infers the correct check of return value, we counted how many allocation functions are inferred to need NULL checking by APISAN. Among 294 allocation functions, APISAN successfully figured out that 164 allocation functions require NULL checking. To assess the accuracy of APISAN's causal relation inference, we counted how many allocation-free functions are inferred as causal relations by APISAN. APISAN found 37 pairs out of 187 such causal relations.

The inaccuracy of APISAN mainly stems from a small number of API usages and limited symbolic execution. For example, if allocated memory is freed by a callback function, APISAN fails to detect the causal relation.

6.6 Performance

Our experiments are conducted on a 32-core Xeon server with 256GB RAM. Constructing a symbolic database for Linux kernel, a one-time task for analysis, takes roughly eight hours and generates 300 GB database. Each checker takes approximately six hours. Thus, APISAN can analyze a large system in a reasonable time bound.

6.7 Our Experience with APISAN

While investigating the bug reports generated by APISAN, we found several interesting bugs, which were introduced while fixing bugs or refactoring code to reduce potential bugs. We believe that it shows that bug fixing is the essential activity during the entire life cycle of any software, and automatic bug finding tools such as APISAN should be scalable enough for them to be integrated into the daily software development process.

Incorrect bug fixes. Interestingly, APISAN found an incorrect bug patch, which was found and patched by KINT [45]. The bug was a missing integer overflow check in ext4 file system, but the added condition was incorrect [8]. Also, the incorrect patch was present for almost four years, showing the difficulty of finding such bugs that can be reproduced only under subtle conditions. Since APISAN gives a higher rank for incorrect condition check for integer overflow, we easily found this bug.

Incorrect refactoring. While investigating PHP integer overflow bugs in Figure 13, we found that the bug was newly introduced when changing string allocation APIs; the new string allocation API, `zend_string_alloc()`, omits an internal integer overflow check, making its callers vulnerable to integer overflow.

7 Discussion

In this section, we discuss the limitations of APISAN's approach and discuss potential future directions to mitigate the limitations.

Limitations. APISAN does not aim to be sound nor complete. In fact, APISAN has false positives (§6.4) as well as false negatives (§6.3, §6.5).

Replacing manual annotations. One practical way to reduce false negatives is to run multiple checkers on the same source code. In this case, APISAN's inference results can be used to provide missing manual annotations required by other checkers. For example, APISAN can provide inferred integer overflow-sensitive APIs to KINT and inferred memory allocation APIs to UC-KLEE.

Interactive ranking and filtering. In our experience, the false positive reports of APISAN are repetitive since incorrect inference of an API can incur many false positive reports. Therefore, we expect that incorporating the human feedback of investigation into APISAN's inference and ranking will significantly reduce false positives and developers' investigation efforts.

Self regression. As we showed in §6.7, bug fixing and refactoring can introduce new bugs. APISAN's approach is also a good fit for self-regression testing by comparing two versions of bug reports and giving higher priorities to changed results.

8 Related Work

In this section, we survey related work in bug finding, API checking, and semantic inference.

Finding bugs. Meta compilation [3, 17, 25] performs static analysis integrated with compilers to enforce domain-specific rules. RacerX [16] proposed flow-sensitive static analysis for finding deadlocks and race conditions. LCLint [20] detects mismatches between source code and user-provided specifications. Sparse [41] is a static analysis tool to find certain types of bugs (e.g., mixing pointers to user and kernel address spaces, and incorrect lock/unlock) in the Linux kernel based on developers' annotations. Model checking has been applied to various domains including file systems [24, 38, 48, 49], device drivers [5], and network protocols [34]. A frequent obstacle in applying these techniques is the need to specify semantic correctness, e.g., domain-specific rules and models. In contrast, APISAN statistically infers semantic correctness from source code; it is generic without requiring models or annotations, but it could incur higher false positives than techniques that use precise semantic correctness information.

Checking API usages. SSLint [26] is a static analysis tool to find misuses of SSL/TLS APIs based on predefined rules. MOPS [9] checks source code against security properties, i.e., rules of safe programming practices. Joern [46] models common vulnerabilities into graph traversals in a code property graph. Unlike these solutions, which are highly specialized for a certain domain (or an API set) and rely on hand-coded rules, APISAN is generally applicable to any domain without manual effort.

Inferring semantics. Engler et al. [18] find deviations from the results of static analysis. Juxta [32] finds deviations by comparing multiple file systems, which follow similar specifications. APISAN's goal is to find deviations in API usages under rich symbolic contexts. DynaMine [30] and VCCFinder [36] automatically extract bug patterns from source code repositories by analyzing bug patches. These approaches would be useful in APISAN as well.

Automatic generation of specifications has been explored by Kremenek et al. [28] for resource allocation, by PRMiner [29] for causal relations, by APIMiner [1] for partial ordering of APIs, by Daikon [19] from dynamic execution traces, by Taghdiri et al. [43] for structural properties, by PRIME [33] for temporal specifications, by Nguyen et al. [35] for preconditions of APIs, by Gruska et al. [23] for sequences of functions, by JIGSAW [44] for resource accesses, by MERLIN [31] for information flow specifications, and by Yamaguchi et al. [47] for taint-style vulnerabilities. These approaches focus on extracting one aspect of the specification. Also, some of them [1, 43] are not scalable because of the complexity of the algorithms used. On the other hand, APISAN focuses on extracting four orthogonal aspects of API usages and using them in combination to find complex bug patterns.

9 Conclusion

We proposed APISAN, a fully automated system for finding API usage bugs by inferring and contrasting semantic beliefs about API usage from source code. We applied APISAN to large, widely-used software, including the Linux kernel, OpenSSL, PHP, and Python, composed of 92 million lines of code. We found 76 previously unknown bugs of which 69 bugs have already been confirmed. Our results show that APISAN's approach is effective in finding new bugs and is general enough to extend easily to custom API checkers based on APISAN.

10 Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This work was supported by DARPA under agreement #15-15-TC-FP-006, #HR0011-16-C-0059 and #FA8750-15-2-0009, NSF awards #CNS-1563848, #DGE-1500084, #1253867 and #1526270, ONR N00014-15-1-2162, ETRI MSIP/IITP[B0101-15-0644], and NRF BSRP/MOE[2015R1A6A3A03019983]. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the 6th joint meeting of European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, Dubrovnik, Croatia, Sept. 2007.

[2] An integer overflow bug in php_str_to_str_ex() led arbitrary code execution. https://bugs.php.net/bug.php?id=71450, 2016.

[3] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (Oakland)*, pages 143–160, Oakland, CA, May 2002.

[4] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the ACM EuroSys Conference*, pages 73–85, Leuven, Belgium, Apr. 2006.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, Alexandria, VA, Oct.–Nov. 2006.

[7] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[8] H. Chen. [PATCH] FS: ext4: fix integer overflow in `alloc_flex_gd()`. http://lists.openwall.net/linux-ext4/2012/02/20/42, 2012.

[9] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, Nov. 2002.

[10] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2002.

[11] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[12] CVE-2014-4113. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-4113, 2014.

[13] CVE-2016-5636. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5636, 2016.

[14] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI'02*, 2002.

[15] M. S. Dittmer and M. V. Tripunitara. The UNIX process identity crisis: A standards-driven approach to setuid. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, Scottsdale, Arizona, Nov. 2014.

[16] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Oct. 2000.

[18] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[19] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, Los Angeles, CA, USA, May 1999.

[20] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the 1994 ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, New Orleans, Louisiana, USA, Dec. 1994.

[21] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17(2), 2008.

[22] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, Raleigh, North Carolina, Oct. 2012.

[23] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 2010 International Symposium on Software Testing and Analysis (ISSTA)*, Trento, Italy, July 2010.

[24] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *Proceedings of the 6th Usenix Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, Feb. 2008.

[25] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of*

the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany, June 2002.

[26] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang. Vetting SSL usage in applications with SSLint. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[27] Heap overflow in zipimporter module. `https://bugs.python.org/issue26171`, 2016.

[28] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[29] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference (ESEC) held jointly with 13th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, Lisbon, Portugal, Sept. 2005.

[30] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference (ESEC) held jointly with 13th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, Lisbon, Portugal, Sept. 2005.

[31] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification Inference for Explicit Information Flow Problems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009.

[32] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[33] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the 2012 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Tucson, AZ, USA, Oct. 2012.

[34] M. S. Musuvathi, D. Park, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[35] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, Hong Kong, Sept. 2014.

[36] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Denver, Colorado, Oct. 2015.

[37] D. A. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *Proceedings of the 24th Usenix Security Symposium (Security)*, Washington, DC, Aug. 2015.

[38] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–280, Dublin, Ireland, June 2009.

[39] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

[40] snd_pcm_new(). `https://www.kernel.org/doc/htmldocs/device-drivers/API-snd-pcm-new.html`, 2016.

[41] Sparse - a Semantic Parser for C. `https://sparse.wiki.kernel.org/index.php/Main_Page`, 2013.

[42] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1), 1986.

[43] M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. In *Proceedings of the 19th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Linz, Austria, Sept. 2004.

[44] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger. JIGSAW: Protecting resource access by inferring programmer expectations. In *Proceedings of the 23rd Usenix Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[45] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[46] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[47] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[48] J. Yang, P. Twohey, and Dawson. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–288, San Francisco, CA, Dec. 2004.

[49] J. Yang, C. Sar, and D. Engler. eXplode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 10–10, Seattle, WA, Nov. 2006.

A Appendix

Function $retvars(e)$ returns all $\langle \mathbf{ret}, i \rangle$ variables in e, which is defined as follows:

$$retvars(e) = \begin{cases} \emptyset & \text{if } e \equiv n \\ \emptyset & \text{if } e \equiv \langle \mathbf{arg}, i \rangle \\ \{\langle \mathbf{ret}, i \rangle\} & \text{if } e \equiv \langle \mathbf{ret}, i \rangle \\ retvars(e') & \text{if } e \equiv uop\ e' \\ retvars(e_1) \cup retvars(e_2) & \text{if } e \equiv e_1\ bop\ e_2 \end{cases}$$

Function $argvars(e,t)$ returns all $\langle \mathbf{arg}, i \rangle$ variables in e, consulting t to recursively replace each $\langle \mathbf{ret}, i \rangle$ variable by its associated function call symbolic expression. It is defined as follows:

$$argvars(e,t) = \begin{cases} \emptyset & \text{if } e \equiv n \\ \{\langle \mathbf{arg}, i \rangle\} & \text{if } e \equiv \langle \mathbf{arg}, i \rangle \\ \bigcup_{j=1}^{|\vec{e}'|} argvars(\vec{e}'[j], t) & \text{if } e \equiv \langle \mathbf{ret}, i \rangle, \text{where} \\ & \quad t[i] \equiv \mathbf{call} * (\vec{e}') \\ argvars(e', t) & \text{if } e \equiv uop\ e' \\ argvars(e_1, t) \cup argvars(e_2, t) & \text{if } e \equiv e_1\ bop\ e_2 \end{cases}$$