Svace Specification Manual
for Svace 3.1

ISP RAS

June 2020
Version 0.5

Content

1 Introduction

This document describes the way for specifying library or user functions behavior for the Svace static analyzer. Such manually created models are called *specifications* in Svace. Svace uses the data derived from specifications to make the analysis more precise. The following chapters introduce Svace analysis workflow, explain the way Svace uses specifications, and describe how they should be written.

1.1 Svace Description

Svace is a static analysis tool for finding errors in programs written in C/C++, Java, and C#. The C# analyzer uses a separate engine that is technically on par with the main engine serving the first three languages. Support for Kotlin and Go languages is underway. Svace is developed in the Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS).

Svace uses the following analysis workflow:

- Perform a controlled program build (monitored by Svace) to capture various build data for further analysis (such as compiler and linker calls, list of compiled files etc.);
- Run lightweight analyzers that typically detect errors as patterns on syntax trees of the correponsing language, or AST-based analyzers (Clang Static Analyzer for C/C++, SpotBugs for Java and some others);
- Run the main path-sensitive and context-sensitive interprocedural analyzer (Svace engine or SvEng);
- Upload results to the Svace history server for storing analysis runs;
- Review results in a web interface backed up by the history server.

Svace engine detects a wide variety of defects including null pointer dereference, memory and resource leaks, buffer overflow, unreachable code, tainted vulnerabilities, leaks of sensitive data, integer overflows, using unitialized data, concurrency errors, division by zero, use of freed memory and others. The engine workflow is as follows:

- building the program call graph;
- running the fast preliminary phase;
- running the main summary-based analysis;
- running the statistical analysis;
- running a number of special analysis passes for C++ constuctors, destructors and assign operators.

Svace engine uses the summary-based approach for performing interprocedural analysis. Such analysis traverses the call graph bottom up starting from leaves, meaning that when recursive cycles are broken, callees are analyzed before callers. When the intraprocedural function analysis pass completes, Svace builds the function summary which contains all information needed for analyzing this function's callers.

Quite often the analyzer finds calls to unknown functions. Also some data may be omitted from a summary because of analysis imprecision or scalability tradeoffs. In such cases Svace can use external models for unknown or not fully precisely analyzed function. These models are written by an analyzer developer or a user and are called specifications. For example, a specification of the `strlen` function can include a note about unconditionally dereferencing its argument. Thus, an error will be reported by Svace when a null pointer is passed to that function, even if its source code is not available.

Svace function specification[1] is just an additional function definition written in the target language source code (C, C++, Java, Kotlin, Go). It describes the needed data about the function behavior in a compact and simple form. Along with the programming language constructs it can use calls to the predefined Svace special functions[2]. Svace analysis engine processes the specifications similarly to other analyzed functions but uses the summary collected from the specifications prior to the functions original definitions.

Svace engine uses function specifications internally for modeling properties of standard libraries. This manual provides many examples from Svace specifications. Most of those examples are simplified versions of real specifications. All details that are irrelevant to the described mechanism are removed for simplicity.

Note: do not add specifications from this manual to Svace as the analyzer already contains their proper extended versions.

1.2 Specifications Overview

A specification is written as a function in the same language and with the same declaration as the modeled function. Svace analyzes the specification source code and uses it when analyzing calls to that function. Specifications allow describing properties of a function, its parameters or the return value. Svace engine will use those properties for modeling program semantics.

For example, the `malloc` specification includes the following code:

```
void *malloc(size_t size) {
    sf_set_trusted_sink_int(size); //size is tainted sink

    void *ptr;
    sf_overwrite(&ptr);  //'ptr' is initialized
    sf_set_alloc_possible_null(ptr, size); //'ptr' may be null
    sf_raw_new(ptr); //pointed memory by 'ptr' is not initialized
```

[1] In other tools they often are called models or annotations.
[2] spec-functions

```
    //size of allocated buffer is 'size'
    sf_set_buf_size(ptr, size);
    return ptr;
}
```

Users can create own specifications via `svace spec` tool (see below).

Important: User specifications take precedence over Svace internal specifications. It is unadvisable to create those for the functions already modeled by Svace as all effects of the Svace internal specification will be lost.

For analyzing specifications Svace first compiles them and then analyzes their source code representation in the same way as the regular functions. This means that specifications need to be correct programs so that they can be compiled without errors.

Svace uses a specification only for modeling a function call. If the project being analyzed contains the modeled function source code (e.g. a library that has own `malloc` implementation), Svace will analyze this function and report possible errors, but **only** its specification will be further used for analyzing calls to that function.

Important: Svace specifications affect all detectors and the engine algorithms. It is not possible to create specifications for a single checker.

2 Tutorial

This chapter contains an example of using Svace function specifications for a simple program. The program uses a custom library for working with files and sending data via network.

2.1 An Example Program

Source code to be analyzed:

```
//file program.c
//a user function that reads data from a file
int get_from_file(int handle);

//a user function that creates a buffer and returns its handle.
//the size parameter 'shouldnt be tainted
int create_my_buffer(int size);

void send_buffer(int bufHandle);

int handle = 0x700;

int read_data_from_file() {
    int data = get_from_file(handle);
    return data;
```

```
}

int main() {
    int val = read_data_from_file();
    //vulnerability: passing tainted data to a secure operation
    int bufHandle = create_my_buffer(val);

    send_buffer(bufHandle);
    return 0;
}
```

The build script:

```
$ cat  build_all.sh
#!/bin/bash
gcc -c program.c
```

Initial Svace run:

```
$ svace init
$ svace build ./build_all.sh
$ svace analyze
```

Svace won't find any vulnerabilities. The reason is that Svace doesn't have any information about functions `create_my_buffer` and `get_from_file`.

2.2 Using Specifications

To provide information about user libraries we have to add Svace specifications. The `lib.c` file with specifications is like following:

```
int get_from_file(int handle) {
    int res;
    sf_overwrite(&res);//note: the & operator is used
    sf_set_tainted_int(res);
    return res;
}

int create_my_buffer(int size) {
    sf_set_trusted_sink_int(size);

    int res;
    sf_overwrite(&res);
    return res;
}
```

Note: Svace matches a specification and a modeled function via function name and argument number.

The below command is used to add the newly written specification to Svace:

```
$ svace spec add lib.c
```

Let us run the analysis again:

```
$ svace analyze
```

This time Svace will find the error like below:

```
TAINTED_INT Integer value 'val' obtained from untrusted source at program.c:13
by calling function 'get_from_file' at program.c:8 without checking
its bounds is used in a trusted operation at program.c:14 by calling
function 'create_my_buffer'.
```

2.3 Fixing The Error

To fix a `TAINTED_INT` warning, we need to change the source so that the value returned from `read_data_from_file` is checked. The updated source looks like below:

```
int get_from_file(int handle);
int create_my_buffer(int size);
void send_buffer(int bufHandle);

int handle = 0x700;

int read_data_from_file() {
    int data = get_from_file(handle);
    return data;
}

#define MAX_SIZE 10000

int main() {
    int val = read_data_from_file();

    if (val > 0 && val <= MAX_SIZE) {
        int bufHandle = create_my_buffer(val);

        send_buffer(bufHandle);
    }
    return 0;
}
```

Since the source code is changed, we have to rebuild the file before the final analysis like below:

```
$ svace build ./build_all.sh
$ svace analyze
```

The error is not reported in this case.

3 C/C++ Specifications

C/C++ specifications are functions written in C/C++. For describing function or parameter properties special Svace functions can be called in the specification. Such functions are called *specfunctions*. Spec functions are prefixed with `sf_` and have a special meaning for Svace.

All public specifications are contained in the `svace/specifications/include/public-specfunc.h` file. Svace internal specifications may use other internal specfunctions that are not advisable for public use.

3.1 Common Specifications

Svace doesn't have a specfunction to denote an argument dereference. It is enough to dereference a variable in the specification source.

For example:

```
void foo(int*a) {
    int val = *a;//parameter a is dereferenced
}

void bar(int**b) {
    int val = **b;//parameter b is dereferenced
                  //and (*b) is also dereferenced.
}
```

If parameter is dereferenced under some condition then the condition can be written as is in the source code:

```
void foo(int*a, int flag) {
    if (flag > 0) {
        int val = *a;//parameter a is dereferenced if flag > 0
    }
}
```

The same technique may be used for other properties. E.g., to denote a buffer access, a specification has to contain a code with buffer access:

```
void access(char* buf, int index) {
    char ch = buf[index];
}
```

It is possible to denote other properties. The exact number of those varies between Svace versions.

A function returns its argument:

```
int ret(char* buf, int index) {
    return index;
}
```

A function returns -1 if argument is negative and 1 in other cases:

```
int ret(int index) {
    if (index < 0) return -1;
    return 1;
}
```

A function fills memory pointed to by a parameter with another parameter:

```
void fill(int* p, int val) {
    *p = val;
}
```

3.2 Execution Terminations

```
void sf_terminate_path(void);
```

The `sf_terminate_path` specfunction denotes that the current execution path is terminated (with calling `exit`-like functions, an infinite loop or crashes).

The specification can be used as follows:

```
void exit(int code) {
    sf_terminate_path();
}

void error(int status, int errnum, const char *fmt, ...) {
    sf_use_format(fmt);

    if (status > 0)
        sf_terminate_path();
}
```

The first specification for `exit` states that the `exit` function never returns. The second specification for `error` states that this function does not return only when the `status` variable is positive.

3.3 Overwrite Specifications

```
void sf_overwrite(void* pval);
void sf_overwrite_int_as_ptr(int pval);
```

Those specifications show that the pointed memory has been assigned with some value. There are two ways of using this specification.

First, it is useful for denoting that a memory reachable via a pointer is overwritten by the modeled function:

```
double modf(double x, double *iptr) {
    sf_overwrite(iptr);
}
```

Now Svace knows that the memory pointed to by `iptr` is assigned with the new value when the `modf` function is called.

The second way of using such a specification is for describing new variables. By default Svace assumes that every variable is not initialized. Therefore the following code is incorrect:

```
double modf(double x, double *iptr) {
    double ret;
    return ret;
}
```

This specification tells Svace that the `modf` function returns an uninitialized value, which was likely not intended. For the correct specification it is needed to add a call to `sf_overwite`:

```
double modf(double x, double *iptr) {
    double ret;
    sf_overwrite(&ret);//note: ‘&‘ is very important here
    return ret;
}
```

Note that `sf_overwrite` takes an address of the memory being initialized and not the memory itself.

Both uses can be combined as below:

```
double modf(double x, double *iptr) {
    sf_overwrite(iptr);

    double ret;
    sf_overwrite(&ret);
    return ret;
}
```

The `sf_overwrite_int_as_ptr` companion function is needed for cases when a pointer is naked (the pointed type is unknown or irrelevant).

For most specfunctions their calling order is not important, but for the `sf_overwrite` functions the order is significant. Changing the calling order will change the specification meaning as shown below.

```
//incorrect specification
ssize_t recv(int s, void *buf, size_t len, int flags) {
    ssize_t res;
    sf_set_possible_negative(res);
    sf_overwrite(&res);
    return res;
}
```

In this example below the uninitilized value of variable `res` is marked as possibly negative. But after the call of `sf_overwrite` the variable is assigned the new value which is unknown (not negative). The correct specifications looks like the following:

```
ssize_t recv(int s, void *buf, size_t len, int flags) {
    ssize_t res;
    sf_overwrite(&res);
    sf_set_possible_negative(res);
    return res;
}
```

3.4 Working with Resources

3.4.1 Acquire and release specfunctions

```
void sf_handle_acquire(void *ptr, int category);
void sf_handle_acquire_int_as_ptr(int ptr, int category);
void sf_handle_release(void *ptr, int category);
```

`sf_handle_acquire` asserts that the `ptr` pointer points to the newly created resource (e.g. a file).

`sf_handle_acquire_int_as_ptr` has the same meaning but is used for resources which are represented in the C code as integers (handles).

`sf_handle_release` asserts that the handle represented by `ptr` was closed or the corresponding resource was released.

The `category` parameter is used for specifying the created resource type. The following values are supported:

```
#define MALLOC_CATEGORY     100
#define KMALLOC_CATEGORY    102
#define VMALLOC_CATEGORY    102
#define NEW_CATEGORY        200
#define NEW_ARRAY_CATEGORY  300
#define BSTR_ALLOC_CATEGORY 400

#define SOCKET_CATEGORY         1200
#define FILE_CATEGORY           1300
```

```
#define HANDLE_FILE_CATEGORY    1400
#define DIR_CATEGORY            1500
#define DL_CATEGORY             1600
#define MMAP_CATEGORY           2000
#define GETADDRINFO_CATEGORY    2100

#define X11_DEVICE              3001
#define X11_CATEGORY            3002

#define SQLITE3_NONFREEABLE_CATEGORY   4000
#define SQLITE3_DB_CATEGORY            4001
#define SQLITE3_MALLOC_CATEGORY        4002
#define SQLITE3_TABLE_CATEGORY         4003
#define SQLITE3_STMT_CATEGORY          4004
#define SQLITE3_BLOB_CATEGORY          4005
#define SQLITE3_VALUE_CATEGORY         4006
#define SQLITE3_MUTEX_CATEGORY         4007
#define SQLITE3_BACKUP_CATEGORY        4008
#define SQLITE3_SNAPSHOT_CATEGORY      4011
```

The values between 0 and 10000 are reserved for internal use in Svace. All user resource categories must have values greater than 10000. The categories are needed for finding errors when a resource is created by one function but released with inappropriate function.

These specifications can be used as below:

```
FILE *fopen(const char *filename, const char *mode) {
    FILE *res;
    sf_overwrite(&res);//res is initialized
    sf_overwrite(res);//pointed memory is also initialized
    sf_handle_acquire(res, FILE_CATEGORY);
    return res;
}

int fclose(FILE *stream) {
    sf_handle_release(stream, FILE_CATEGORY);
}
```

For working with memory Svace has the separate specfunction set:

```
void sf_new(void* pval, int category);
void sf_delete(void* pval, int category);
void sf_strdup_res(void* pval);
```

Svace has two types of warnings for resource leaks. The first is called `MEMORY_LEAK` and the second is called `HANDLE_LEAK`. The above functions have the same meaning as their handle counterparts. However, when Svace finds a resource leak, with those specifications it will emit a warning of type `MEMORY_LEAK`

instead of HANDLE_LEAK. Also, Svace handles null values differently: if a pointer is null than the memory is not acquired; for HANDLE_LEAK groups zero handles are assumed being legal handles for acquired resources.

Svace has special warnings type for the strdup function return value. For denoting that a function returns a strdup result, Svace has the sf_strdup_res specfunction. It has the same semantic as sf_new.

Below is an example of user specifications that work with special resources:

```
#define WINDOW_HANDLE_CATEGORY 20001
#define DB_HANDLE_CATEGORY 20002

//create new window
int createWindow(const char *id) {
    int res;
    sf_overwrite(&res);
    sf_overwrite(res);
    sf_handle_acquire(res, WINDOW_HANDLE_CATEGORY);
    return res;
}

//destroy window with the given handle
void closeWindow(int handle) {
    sf_handle_release(handle, WINDOW_HANDLE_CATEGORY);
}

//create new data base connection
int createDB(const char *path, const char*login) {
    int res;
    sf_overwrite(&res);
    sf_overwrite(res);
    sf_handle_acquire(res, DB_HANDLE_CATEGORY);
    return res;
}

//destroy connection with the given handle
void closeDB(int handle) {
    sf_handle_release(handle, DB_HANDLE_CATEGORY);
}
```

Now Svace knows that functions createWindow and createDB create resources of corresponding types and the closeWindow and closeDB functions release them. As a result, in the following code Svace finds a resource leak:

```
int createIf(const char *id, const char* flags) {
    int handle = createWindow(id);

    if (flags < 0)//here is error: closeWindow is not called
```

```
        return ERROR;

    return handle;
}
```

For the following code Svace detects that a resource is not properly released:

```
void readDB(const char *id) {
    int handle = createDB(id, "admin");

    readDBImpl(handle);

    closeWindow(handle);//error - handle is not for window resources.
}
```

3.4.2 Conditional acquire

```
void sf_not_acquire_if_eq(const void* pval, int var, int code);
void sf_not_acquire_if_eq_int_as_ptr(int handle, int var, int code);
void sf_not_acquire_if_less(const void* ptr, int var, int val);
void sf_not_acquire_if_less_int_as_ptr(int handle, int var, int val);
void sf_not_acquire_if_greater(const void* ptr, int var, int val);
void sf_not_acquire_if_greater_int_as_ptr(int handle, int var, int val);
```

Many resource acquire functions return an error code when a memory or a resource were not created. The above functions are used for modeling such situations.

The `sf_not_acquire_if_eq` specfunction marks that the memory pointed to by `pval` was not created if $var = code$. The `sf_not_acquire_if_eq_int_as_ptr` specfunction does the same for handles.

The `sf_not_acquire_if_less` specfunction marks that the memory pointed to by `pval` was not created if $var < val$. The `sf_not_acquire_if_less_int_as_ptr` specfunction does the same but handles.

The `sf_not_acquire_if_greater` specfunction marks that the memory pointed to by `pval` was not created if $var > val$. The `sf_not_acquire_if_greater_int_as_ptr` specfunction does the same for handles.

The examples of using these functions are below.

```
FILE *fopen(const char *filename, const char *mode) {
    FILE *res;
    sf_overwrite(&res);
    sf_overwrite(res);
    sf_handle_acquire(res, FILE_CATEGORY);
    sf_not_acquire_if_eq(res, res, 0);
    return res;
}
```

This specification says that the `fopen` function creates a resource and this resource is not created if it equals to null. Note that calling `sf_handle_acquire` is still needed: the call of `sf_not_acquire_if_eq` doesn't state that a resource is created.

```
int asprintf(char **ret, const char *format, ...) {
    sf_overwrite(ret);
    sf_overwrite(*ret);
    sf_new(*ret, MALLOC_CATEGORY);

    int res;
    sf_not_acquire_if_less(*ret, res, 1);
    return res;
}
```

For the `asprintf` function specification Svace gets information that the new memory is created and the pointer to this memory is written to the memory pointed to by the first parameter. In case of failure the return value will be less that 1.

```
int open(const char *name, int flags, ...) {
    int x;
    sf_overwrite(&x);
    sf_overwrite_int_as_ptr(x);
    sf_handle_acquire_int_as_ptr(x, HANDLE_FILE_CATEGORY);
    sf_not_acquire_if_less_int_as_ptr(x, x, 3);
    return x;
}
```

This specification says that the `open` function creates the new resource. If it is less than 3 than it was not created. The original `open` function returns -1 in case of error, but in real code it is common to check the return value to be positive. That's why we didn't use the `sf_not_acquire_if_eq_int_as_ptr` specfunction. 0, 1, and 2 are used for `stdin`, `stdout`, and `stderr`, respectively. It is not necessary to release those handles. That's why the last argument of `sf_not_acquire_if_less_int_as_ptr` is 3.

3.4.3 Memory management specifications

```
void sf_invalid_pointer(void *newp, void *p);
void sf_escape(const void* ptr);
```

The `sf_invalid_pointer` specfunction marks that its second argument `p` is invalid when it is not equal to the first argument, `newp`. It is used for modeling the `realloc` function:

```
void *realloc(void *ptr, size_t size) {
    void *retptr;
```

```
    sf_overwrite(&retptr);
    sf_overwrite(retptr);
    sf_new(retptr, MALLOC_CATEGORY);
    sf_invalid_pointer(ptr, retptr);
    return retptr;
}
```

From this specification Svace knows that `realloc` creates the new memory and returns a pointer to it. If the new memory is not equal to the first argument, than the result is invalid and cannot be used.

The `sf_escape` specfunction denotes that the memory pointed to by `pval` is escaped. Now the pointer itself and all referenced values (the memory reachable via this pointer) can be saved in a variable or deleted. Svace won't emit memory leak warnings for all these values.

```
int putenv(char *cmd) {
    sf_escape(cmd);
}
```

The `putenv` function stores its parameter internally. So for the following code Svace does not emit leak warnings:

```
putenv(strdup(var));
```

3.5 Tainted Specifications

Most tainted checkers are written as source-sink checkers. A *source* is a function that gets its data from external environment (a file, user input, network and etc.). A *sink* is a vulnerable operation where the source data should not be used. *Sanitization* is a process of checking or correcting the source data so it becomes safe for further usage.

Svace always treats array accesses as vulnerable operations for tainted integer indexes.

3.5.1 Sources of tainted values

```
void sf_set_tainted(const void* str);
void sf_set_tainted_int(int i);
void sf_set_tainted_char(int i);
void sf_set_tainted_interval(int i, int low, int upper);
void sf_set_tainted_buf(void *ptr, int size, int shift);
```

The `sf_set_tainted` specfunction marks its string argument as tainted which means the following:

- The string length may be controlled by an attacker.

- The string content may be filled by an attacker. Values created from this string will be marked as tainted too.

The example usage is shown below.

```
char *gets(char *s) {
    sf_overwrite(s);
    sf_set_tainted(s);

    char *str;
    sf_overwrite(&str);
    sf_set_tainted(str);
    return str;
}
```

The `sf_set_tainted_int` specfunction indicates that its integer argument is tainted. It may have any values. The `sf_set_tainted_char` specfunction has the same meaning for the 8-bit character type.

The example is below.

```
uint32_t htonl(uint32_t hostlong) {
    uint32_t res;
    sf_overwrite(&res);
    sf_set_tainted_int(res);
    return res;
}
```

The `sf_set_tainted_interval` specfunction indicates that its first parameter is tainted and has values inside the $[low; upper]$ interval. So $i \geq low$ and $l \leq upper$.

For the below example the `fgets` specification shows that the function returns a character value or -1 (in case of error).

```
int fgetc(FILE *stream) {
    int res;
    sf_overwrite(&res);
    sf_set_tainted_interval(res, -1, 255);
    return res;
}
```

The `sf_set_tainted_buf` specfunction marks the memory pointed to by `ptr` as tainted. The memory size (number of tainted bytes) is (`size` + `shift`). If `size` is zero that there is no limit. The `shift` parameter is a constant difference between the size limit and the maximum string length (0 for most functions, -1 for `fgets`).

The example usage is below.

```
char *fgets(char *s, int num, FILE *stream) {
    sf_overwrite(s);
```

```
    sf_set_tainted(s);
    sf_set_tainted_buf(s, num, -1);

    char *str = s;
    return str;
}
```

In this specification Svace treats `num` bytes pointed to by the modeled function return value as tainted.

3.5.2 Converting functions

```
void sf_str_to_int(const void* str, int i);
void sf_str_to_long(const void* str, long i);
```

The `sf_str_to_int` specfunction indicates that `i` is an integer representation of `str` string content. The `sf_str_to_long` specfunction does the same for long values.

Currently Svace uses a heuristic that if a program converts a string to integer, then the conversion result is tainted. In most cases arguments of those functions are tainted.

The example usage is below.

```
int atoi(const char *arg) {
    int res;
    sf_overwrite(&res);
    sf_str_to_int(arg, res);
    return res;
}
```

3.5.3 Transferring taintedness

```
void sf_transfer_tainted(void* dst, void* src, int len);
```

The `sf_transfer_tainted` specfunction marks `len` bytes of the first parameter, `dst`, as tainted when the second parameter, `src`, is tainted. The example usage is below.

```
void *memcpy(void *dst, const void *src, size_t num) {
    sf_transfer_tainted(dst, src, num);

    return dst;
}
```

3.5.4 Sinks for tainted values

```
void sf_use_format(const void* str);
void sf_set_trusted_sink_int(int i);
void sf_set_trusted_sink_char(int i);
void sf_set_trusted_sink_ptr(const void* str);
```

The `sf_set_trusted_sink_int` specfunction indicates that the `i` parameter is used in a vulnerable operation. Its bounds have to be checked for some values. It is possible to check values by comparing with other variable.

Consider the example code below for detailed usage.

```
int gettainted() {
    int tainted_res;
    sf_overwrite(&tainted_res);
    sf_set_tainted_int(tainted_res);
    return tainted_res;
}

int trusted(int c) {
    sf_set_trusted_sink_int(c);
}

void error() {
    int index = gettainted();

    //Svace emits a warning that 'index' bounds are not checked
    trusted(index);
}

void errorToo() {
    int index = gettainted();

    if (index > 10)
        return;

    //Svace emits a warning that the lower bound is not checked
    trusted(index);
}

void noError() {
    int index = gettainted();

    if (index < 0 || index > 10)
        return;
```

```
    //no warnings, both lower and upper bounds are checked
    trusted(index);
}

void compareWithVar(int limit) {
    int index = gettainted();

    if (index < 0 || index > limit)
        return;

    //no warnings, both lower and upper bounds are checked
    trusted(index);
}
```

Another example of using `sf_set_trusted_sink_int` is in the `malloc` specification:

```
void *malloc(size_t size) {
    sf_set_trusted_sink_int(size);
}
```

This code says that the `malloc` parameter that comes from external sources has to be checked for bounds. Svace doesn't find exact bound limits because they are application specific.

The `sf_set_trusted_sink_char` specfunction has the same meaning for character type. The only difference that instead of emitting a warning of type `TAINTED_INT` Svace will emit its subtype called `TAINTED_INT.CTYPE`.

The `sf_set_trusted_sink_ptr` specfunction states that its string argument is used in a vulnerable operation. It shouldn't get tainted strings.

The example is below.

```
int execl(const char *path, const char *arg0, ...) {
    sf_tocttou_access(path);
    sf_set_trusted_sink_ptr(path);
    sf_fun_does_not_update_vargs(2);
    // The exec* functions return only if an error has occurred.
    int res;
    sf_set_errno_if(res, sf_top());
    sf_no_errno_if(res, sf_bottom());
    return res;
}
```

The `sf_use_format` specfunction indicates that its argument is a `printf`-like format string. It shouldn't get tainted arguments.

The example is below.

```
int fprintf(FILE *stream, const char *format, ...) {
    sf_use_format(format);
```

```
}
```

3.5.5 Sanitization

```
void sf_sanitize(const char* str);
```

The `sf_sanitize` specfunction removes the tainted mark from its arguments. It can be used in the following cases:

- For functions that do not return if their arguments are not proper:

```
void checkArg(const char *s) {
    int len = strlen(s);
    if (len > 10)
        exit(1);

    for (int i=0; i < len; ++i) {
        if(isBad(s[i])
            exit(1);
    }
}
```

- For functions with complex logic that check arguments for taintedness.

```
int strcmp(const char *s1, const char *s2) {
    sf_sanitize(s1);
    sf_sanitize(s2);
}
```

The above specification just removes the tainted mark from `strcmp` arguments as Svace considers this call an evidence of sanitization.

3.6 Buffers and Strings

3.6.1 Buffers

```
void sf_buf_size_limit(const void* ptr, int bytes_size);
void sf_buf_size_limit_strict(const void* ptr, int bytes_size);
void sf_buf_size_limit_read(const void* ptr, int bytes_size);
```

The `sf_buf_size_limit` specfunction shows that the `ptr` parameter may be used to access a buffer with offsets $[0, size)$. Unless `sf_buf_stop_at_null` (see 3.6.2) is used, the access doesn't stop at null terminators.

The `sf_buf_size_limit_read` specfunction has the similar meaning as `sf_buf_size_limit` but the buffer access is read-only.

The examples are shown below.

```
char *fgets(char *s, int num, FILE *stream) {
    sf_overwrite(s);
    sf_buf_size_limit(s, num);

    char *str = s;
    return str;
}

int snprintf(char *str, size_t size, const char *format, ...) {
    sf_bitinit(str);
    sf_buf_size_limit(str, size);
}
```

A null-terminated example is like following:

```
char *strncat(char *s, const char *append, size_t count) {
    sf_buf_size_limit_read(append, count);
    sf_buf_stop_at_null(append);
}
```

The specification says that the `strncat` function stops reading the `append` string where it contains null.

The `sf_buf_size_limit_strict` specfunction specifies that limit parameter must be withing buffer bounds. If Svace defines size of buffer and won't be able to define value of parameter than the warning is emitted.

Example:

```
errno_t memcpy_s(void *dst, size_t dstSize, const void *src, size_t num) {
    sf_buf_size_limit_strict(dst, dstSize);
    sf_buf_size_limit(src, num);

    errno_t res;
    sf_overwrite(&res);
    return res;
}
```

The specification above says that parameter `src` is limited by parameter `num` and parameter `dst` is limited by parameter `dstSize`. The difference is in Svace behavior for cases where buffer size is known and parameter values are not known. In this case Svace emits warning only for parameter `dst`. For parameter `src` a warning will be emitted only if Svace defined that parameter's value leads to overflow.

3.6.2 Nonterminated strings

```
void sf_buf_stop_at_null(const void* dst);
void sf_set_possible_nnts(void* dst);
```

The `sf_buf_stop_at_null` specfunction says that the pointer is used to access a string until a null byte is encountered. Unless `sf_buf_size_limit` (see 3.6.1) is used, the access stops only at a null byte.

The example can be seen when specifying the `atoi` function below.

```
int atoi(const char *arg) {
    sf_buf_stop_at_null(arg);
}
```

The `sf_set_possible_nnts` specfunction shows that `dst` can become a non-terminated string. Such values are handled by Svace as a source of non-terminated string errors.

The example is the `recv` function specification as below.

```
ssize_t recv(int s, void *buf, size_t len, int flags) {
    sf_overwrite(buf);
    sf_set_possible_nnts(buf);
    sf_bitinit(buf);
}
```

3.6.3 Modeling `strlen` functions

```
void sf_strlen(size_t len, const char* str);
void sf_wcslen(size_t res, const wchar_t *s);
```

The `sf_strlen` specfunction denotes that `len`, the first parameter, is a length of the C string `str`. The `sf_wcslen` specfunction does the same for wide strings.

The example of specifying the `strlen` function follows.

```
size_t strlen(const char *s) {
    size_t res;
    sf_overwrite(&res);
    sf_strlen(res, s);
    return res;
}
```

3.7 Value properties

3.7.1 Negative values

```
void sf_set_possible_negative(int int_val);
void sf_set_must_be_positive(int int_val) ;
```

The `sf_set_possible_negative` specfunction states that its parameter may have a negative value. It has to be checked before accessing buffers and using in functions that don't expect negative values.

```
int putchar(int c) {
    int ret;
    sf_overwrite(&ret);
    sf_set_possible_negative(ret);
    return ret;
}
```

The `sf_set_must_be_positive` specfunction states that its parameter can't be negative. Zero is accepted.

```
int dup(int oldd) {
    sf_set_must_be_positive(oldd);
}
```

3.7.2 Function results

```
void sf_must_be_checked(int var);

void sf_fread(int res, int file);
void sf_ferror(int var);
void sf_feof(int var);
```

The `sf_must_be_checked` specfunction specifies that its argument must be somehow checked by caller function. The argument denotes some error code. Svace dosn't have any requirements to the checking. It may be compared with any other value.

The example below is taken from the `munmap` function specification.

```
int munmap(void *addr, size_t len) {
    int res;
    sf_overwrite(&res);
    sf_must_be_checked(res);
    return res;
}
```

The `sf_fread`, `sf_ferror`, `sf_feof` specfunctions are needed for the `UNCHECKED_FUNC_RES.FREAD` checker. The checker emits a warning if the return value of `fread`-like functions was checked for error but the `ferror` and `feof` functions were not called.

The examples of specifying this function family is below.

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream) {
    size_t res;
    sf_overwrite(&res);
    sf_must_be_checked(res);
    sf_fread(res, stream);
```

```
    return res;
}

int feof(FILE *stream) {
    int res;
    sf_overwrite(&res);
    sf_feof(stream);
    return res;
}

int ferror(FILE *stream) {
    sf_ferror(stream);
}
```

The simple error code example is below.

```
int foo(FILE *pFile, long lSize, char* buffer) {
    int res = fread(buffer, 1, lSize, pFile);
    if (!res) {//error: no ferror and feof
        return 1;
    }
    return 0;
}

int correct(FILE *pFile, long lSize, char* buffer) {
    int res = fread(buffer, 1, lSize, pFile);
    if (!res) {
        if (ferror(pFile))
            printf ("Error!\n");
        return 1;
    }
    return 0;
}
```

3.8 Other Specifications

3.8.1 Initialization

```
void sf_bitcopy(void* dst, const void* src);
void sf_bitinit(void* ptr);
void sf_deepinit(void* ptr);
int sf_get_some_int();
```

The `sf_bitcopy` specfunction shows that the memory pointed to by `dst` is initialized with the content of `src`.

Example:

```
void *memcpy(void *dst, const void *src, size_t num) {
    sf_bitcopy(dst, src);
}
```

The `sf_bitinit` specfunction marks values pointed to by `ptr` as completely initialized. The `sf_deepinit` specfunction does the same for *all* memory that is reachable from `ptr`. If `ptr` points to a structure, then all its fields are initialized. If a field is a structure itself or points to a structure, then those fields are also initialized.

The examples are shown below.

```
void *memset(void *ptr, int value, size_t num) {
    sf_bitinit(ptr);
}

ssize_t recvmsg(int s, struct msghdr *msg, int flags) {
    sf_deepinit(msg);
}
```

With this specification, the `recvmsg` function initializes not only the `msg` parameter itself but also its fields.

Important: do not confuse `sf_bitinit` and `sf_overwrite` (3.3) functions. `sf_overwrite` shows that the variable changes its value to a completely new one. As a result, the analysis will stop tracking all the previous properties of this variable. The `sf_bitinit` specfunction only marks the variable as being initialized. It does not affect any other properties, which will remain the same.

The below examples illustrate the differences between the two specfunctions.

```
void hard_init(void*p) {
    sf_overwrite(p);
}

void soft_init(void*p) {
    sf_bitinit(p);
}

int foo() {
    char** p = malloc(sizeof(char*));
    *p = malloc(sizeof(char));
    hard_init(p);
    int ret = **p;
    free(*p);
    free(p);
    return ret;
}
```

```
int bar() {
    char** p = malloc(sizeof(char*));
    *p = malloc(sizeof(char));
    soft_init(p);
    int ret = **p;
    free(*p);
    free(p);
    return ret;
}
```

For the `foo` function Svace emits a memory leak warning because the value of the `'p'` pointer that pointed to `headp` was overwritten and lost.

The `sf_get_some_int` specfunction creates the newly initialized integer.

```
int sqlite3_close(sqlite3 *db) {
    return sf_get_some_int();
}
```

This function may be used instead of `sf_overwrite` for new variables:

```
int sqlite3_close(sqlite3 *db) {
    int res;
    sf_overwrite(&res);
    return res;
}
```

3.8.2 Null pointer dereference

```
void sf_set_possible_null(const void* ptr);
void sf_not_null(const void* ptr);
```

The `sf_set_possible_null` specfunction marks the pointer as containing null value. It should be checked for being null before dereferencing. The `sf_not_null` specfunction marks the pointer as non-null. Svace won't emit dereference warnings for this pointer.

The examples for these functions are shown below.

```
jbooleanArray NewBooleanArray(JNIEnv *env, jsize length) {
    jobject *res;
    sf_overwrite(&res);
    sf_set_possible_null(res);
    return res;
}
```

Function `NewBooleanArray` may return null. Its result has to be checked before dereferencing.

Svace doesn't have a special function for denoting dereferences. As mentioned in Section 3.1, to write a specification for a function that dereferences its argument it is enough to add the dereference to the code:

```
FILE *fdopen(int fildes, const char * mode) {
    char d1 = *mode; //dereference here
}
```

3.8.3 Lock-unlock

```
void sf_lock(const void* lock);
void sf_unlock(const void* lock);
void sf_trylock(const void* lock);
void sf_thread_shared(const void *data);
void sf_success_lock_if_zero(int code, const void* lock);
```

The `sf_lock` specfunction marks its argument as a locked mutex. The `sf_unlock` specfunction marks its argument as an unlocked mutex. The `sf_trylock` specfunction marks its argument as a locked mutex but it doesn't lead to the `DOUBLE_LOCK` warning. The mutex will be locked only if it was in the unlocked state. Otherwise nothing will be changed.

These specfunctions are used when writing locking function specifications as shown below.

```
typedef struct pthread_mutex pthread_mutex_t;
struct pthread_mutex;

int pthread_mutex_lock(pthread_mutex_t *mutex) {
    sf_lock(mutex);
}

int pthread_mutex_unlock(pthread_mutex_t *mutex) {
    sf_unlock(mutex);
}

int pthread_mutex_trylock(pthread_mutex_t *mutex) {
    sf_trylock(mutex);
}
```

The `sf_thread_shared` specfunction indicates that the `data` parameter is used by another thread, as shown below.

```
struct pthread;
typedef struct pthread pthread_t;

struct pthread_attr;
typedef struct pthread_attr pthread_attr_t;
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg) {
    sf_thread_shared(arg);
}
```

The `sf_success_lock_if_zero` specfunction associates a variable with the mutex state. If the variable is not zero, then the mutex was not locked. The example can be seen in specifying the `pthread_mutex_lock` function.

```
int pthread_mutex_lock(pthread_mutex_t *mutex) {
    sf_lock(mutex);

    int res;
    sf_overwrite(&res);
    sf_success_lock_if_zero(res, mutex);
    return res;
}
```

3.8.4 Other concurrency specifications

```
void sf_long_time();
```

The `sf_long_time` specification marks the caller function as a function which may be executed for a long time. Locking such function may lead to the performance degradation. The examples can be seen below.

```
int listen(int sockfd, int backlog) {
    sf_long_time();

    int res;
    sf_overwrite(&res);
    sf_set_possible_negative(res);
    return res;
}

unsigned int sleep(unsigned int ms) {
    sf_long_time();
}
```

3.8.5 Variable argument functions

```
void sf_fun_does_not_update_vargs(int from);
void sf_fun_updates_vargs(int from);
```

The `sf_fun_does_not_update_vargs` and `sf_fun_updates_vargs` specfunctions are needed for modeling functions with variable arguments number. The `sf_fun_does_not_update_vargs` specfunction states that the caller function doesn't update its arguments starting from `from` meaning that the following arguments are read only. Its counterpart `sf_fun_updates_vargs` states that the caller function changes its arguments from the specified number `from`, so after calling such a function the passed arguments are initialized.

Two examples with `fprintf` and `fscanf` functions are below.

```
int fprintf(FILE *stream, const char *format, ...) {
    sf_fun_does_not_update_vargs(2);
}

int fscanf(FILE *stream, const char *format, ...) {
    char derefStream = *((char *)stream);
    char d1 = *format;
    sf_use_format(format);

    sf_fun_updates_vargs(2);
}
```

3.8.6 Vulnerable functions

```
void sf_vulnerable_fun(const char*const reason);
void sf_vulnerable_fun_temp(const char*const reason);
void sf_vulnerable_fun_type(const char*const reason,
                            const char*const type);
void sf_vulnerable_fun_sscanf(const char*const reason);
void sf_fun_rand();
```

The above specfunctions are used when there is a need to deprecate some functions. For most of them Svace doesn't do any complicated analysis but just emits a warning when the modeled function was called.

The `sf_vulnerable_fun` specfunction marks the caller function as vulnerable which shouldn't be used. This specfunction may be used for any deprecated function. For those functions Svace emits the `PROC_USE.VULNERABLE` warning. The `sf_vulnerable_fun_temp` specfunction does the same but the emitted warning type is `PROC_USE.VULNERABLE.TEMP`, as shown below.

```
char *strcat(char *s, const char *append) {
    sf_vulnerable_fun("This␣function␣is␣unsafe,"
        "␣use␣strncat␣instead.");
}

char *tempnam(const char *dir, const char *pfx) {
    sf_vulnerable_fun_temp("This␣function␣is␣susceptible␣"
```

```
    "to␣a␣race␣condition␣occurring␣between␣"
    "selection␣of␣the␣file␣name␣and␣creation␣"
    "of␣the␣file,␣which␣allows␣malicious␣"
    "users␣to␣potentially␣overwrite␣"
    "arbitrary␣files␣in␣the␣system.␣"
    "Use␣mkstemp(),␣mkstemps(),␣or␣mkdtemp()␣instead.");
}
```

The `sf_fun_rand` specfunction is the same as `sf_vulnerable_fun` but Svace emits the `PROC_USE.RAND` warning.

The `sf_fun_rand` specfunction is the same as `sf_vulnerable_fun_sscanf` but Svace emits the `PROC_USE.VULNERABLE.SSCANF` warning, as shown below.

```
int rand(void) {
    int res;
    sf_overwrite(&res);
    sf_set_values(res, 0, 32767);//use RAND_MAX?
    sf_fun_rand();
    return res;
}
```

The `sf_vulnerable_fun_type` specfunction allows setting the suffix for the emitted warning, as shown below for the `getenv` function specification.

```
char *getenv(const char *key) {
    sf_vulnerable_fun_type("getenv␣env␣value␣can␣be␣hooked␣by␣a␣hacker",
                           "GETENV");
}
```

Now when the call to `getenv` is detected, Svace emits the `PROC_USE.VULNERABLE.GETENV` warning.

Note: for some cases where Svace can discover that the function call is safe it won't emit the warning. For example:

```
void foo(char*buf) {
    int x;
    char str[10];
    sscanf(buf, "%d%5s", &x, str);
}
```

In this code the `sscanf` functiion fills only 5 bytes in the `str` string. Since its size is 10 it won't lead to any errors.

3.8.7 `sprintf`, `scanf`-like functions

```
void sf_fun_sprintf_like();
void sf_fun_scanf_like(int format_string_index);
void sf_fun_printf_like(int format_string_index);
```

The `sf_fun_sprintf_like` specfunction denotes that the caller function is a `sprintf`-like function meaning that it behaves in a similar fashion.

The `sf_fun_scanf_like` specfunction denotes that the caller function is a `scanf`-like function. Its parameter is the format string argument index (starting from 0).

The `sf_fun_printf_like` specfunction denotes that the caller function is a `printf`-like function. Its parameter is the format string argument index (starting from 0).

The examples are shown below.

```
int sprintf(char *s, const char *format, ...) {
    sf_fun_sprintf_like();
}

int sscanf(const char *s, const char *format, ...) {
    sf_fun_scanf_like(1);
    sf_fun_updates_vargs(2);
}

int fprintf(FILE *stream, const char *format, ...) {
    sf_fun_printf_like(1);
    sf_fun_does_not_update_vargs(2);
}
```

3.8.8 Sensitive data leaks

```
void sf_sec_leak(const void*data, size_t size);
```

The `sf_sec_leak` specfunction says that the data from the `data` buffer of size `size` is saved to the external memory and may be leaked.

3.8.9 Exceptions

```
void sf_could_throw(const char*);
void sf_could_throw_pedantic(const char*);
```

The `sf_could_throw` specfunction indicates that the caller function may throw an exception. This exception should be caught in the analyzed program. The `sf_could_throw_pedantic` specfunction has the similar meaning but if the program doesn't catch an exception, Svace emits warning with lower priority. Both functions have an argument, which is the exception name. Svace will compare the name with the names of types used in catch statements.

3.9 Combining Values

```
int sf_range(int left, int right);
int sf_cond_range(const char *cond, int right);
int sf_cond(int left, const char *cond, int right);
int sf_join(int v1, ...);
int sf_meet(int v1, ...);
```

The above specfunctions offer a flexible way of specifying restrictions on integer values as well as other conditions. All functions return a value which has the specified properties. It may be used in other specifications.

The `sf_range` specfunction creates an integer variable which value lies inside the $[left; right]$ range, as in the example below.

```
int x = sf_range(1, 10);
//Now variable x has values between 1 and 10.
```

The `sf_cond_range` specfunction creates an integer variable which has the value restricted by a condition. Its first argument can have the following values: `">"`, `"<"`, `">="`, `"<="`, and `"=="`. The example is shown below.

```
int x = sf_cond_range(">", 7);
//Now variable x has values more than 7
```

The `sf_cond` specfunction associates a condition with the return value as follows: `left cond right` where `cond` can have the following values: `">"`, `"<"`, `">="`, `"<="`, `"=="`, and `"!="`. `Left` and `right` can be either constants and variables. In the below specification `sf_cond(a, ">", 0)` associates the $a > 0$ condition with the return value stored in `cond`.

```
void foo(int a, int b, int c) {
    int cond = sf_cond(a, ">", 0);
    if (cond)
        return b;
    return c;
}
```

The `sf_join` and `sf_meet` specfunctions allow combining results of other functions listed here: `sf_join` joins values of its arguments, while `sf_meet` intersects values of its arguments.

The example for these functions is shown below.

```
void foo(int a, int b, int c) {
    int c1 = sf_cond(a, ">", 0);
    int c2 = sf_cond(b, ">=", c);
    int c3 = sf_cond(c, "<", 10);

    int join = sf_join(c1, c2);
    int meet = sf_meet(join, c3);
    if (meet)
```

```
        return 0;
    return c;
}
```

4 Java specifications

TBD

5 Kotlin specifications

TBD

6 Go specifications

TBD

7 Specifications in comments

TBD

Index of Specifications in Alphabetical Order

Index of Specifications by Sections