Sysfilter+
Type-based System Call Filtering with Temporal Specialization

Richard Abou Chaaya
Brown University

Abstract
Many program analysis tools rely on the program’s call graph. This is the case of Sysfilter [3], which constructs a complete call graph of the target program to obtain a list of the system calls that the program requires. However, indirect calls are a major challenge when constructing a precise call graph. This is because determining where a given function pointer will point to at runtime is, in general, undecidable. As a result, the set of potential indirect targets needs to be largely over-approximated, which hinders the precision of static analysis tools. We study how type information extracted from debug symbols can be used to refine the static analysis and call graph construction for binary-only targets.

We introduce Sysfilter+, which extends Sysfilter to use type information during its analysis. We also study the impact this has on temporal system call filtering and implement a temporal system call filtering enforcement strategy for threads to validate our method. We conclude with a large-scale analysis of 30K binaries on Debian Sid to observe the impact type information has on the constructed call-graphs and how this translates to system call reduction. Using type information we reduce the number of potential indirect targets by 35% on average and the number of syscalls by 4%.

1 Introduction
Sysfilter [3] is a binary analysis framework that automatically determines a safe over-approximation of the set of system services used by a program. It then restricts the program to only using those syscalls in accordance with the principle of least privilege.

Sysfilter relies on static analysis in order to obtain a complete set of system calls. However, indirect calls are hard to analyze statically as knowing where a given function pointer will point to at runtime is, in general, undecidable. This forces Sysfilter to consider every address-taken function as a potential indirect call target which leads to over-approximating the call-graph.

We introduce Sysfilter+, which extends Sysfilter to use type information obtained from debug symbols to reduce the set of potential indirect targets. Furthermore, by starting the type analysis at different points in the program, we also analyze its effect on temporal syscall specialization.

2 Background on DW ARF
This section contains an introduction to the DWARF [2] features relevant to this project. We use debug information to obtain the types of functions and the types of global and local variables. This information is present in the .debug_info section of ELF binaries compiled with debug symbols. In the .debug_info section, each compilation unit has a header followed by Debugging Information Entries (DIEs).

2.1 Debugging information entries
Each DIE contains a tag which indicates its type and attributes which provide additional information. These attributes may in turn refer to other DIEs. Finally, DIEs can have children entries. For example, a DW_TAG_subroutine DIE (which represents a function) will have children entries such as DW_TAG_formal_parameter to represent an argument and DW_TAG_variable to represent a local variable.

2.2 Example
Consider the following code

```c
1  int my_global_int = 1;
2  int my_func(char* arg) {
3      int my_int = 0;
4      return 0;
5  }
```

Here is the corresponding .debug_info section (some attributes and entries have been removed for brevity). The
rest of this section describes how it can be used to obtain the
types of variables and functions.

### Content of the .debug_info section:

1. **Compilation Unit** @ offset 0x0:
   - `<0><b>`: (DW_TAG_compile_unit)
   - `<2d>`: (DW_TAG_variable)
     - `DW_AT_name`: `my_global_int`
     - `DW_AT_type`: `<0x43>`
   - `<43>`: (DW_TAG_base_type)
     - `DW_AT_byte_size`: 4
     - `DW_AT_name`: `int`
   - `<9b>`: (DW_TAG_pointer_type)
     - `DW_AT_byte_size`: 8
     - `DW_AT_type`: `<0xa0>`
   - `<9a>`: (DW_TAG_formal_parameter)
     - `DW_AT_name`: `arg`
     - `DW_AT_type`: `<0xa0>`
   - `<da>`: (DW_TAG_variable)
     - `DW_AT_name`: `my_int`
     - `DW_AT_type`: `<0x43>`

• The DIE at offset `2d` represents a variable (DW_TAG_variable). Since it is at depth 1, it is a global variable. The DW_AT_name attribute contains its name and the DW_AT_type attribute contains the offset of the DIE that represents its type. So looking at the DIE at offset `43`, we can see that this is an int.

• The DIE at offset `cb` represents a function (DW_TAG_subprogram). Its DW_AT_type attribute is the function’s return type. It has two children DIEs, a DW_TAG_formal_parameter and a DW_TAG_variable which represent the function’s argument and local variable respectively. The type of the parameter is the DIE at offset `9a`. We can see that it’s a pointer (DW_TAG_pointer_type) and its DW_AT_type attribute is the DIE at offset `a6` which is the entry representing the char type. This means that the function’s argument is a pointer to a char.

### 3 Type Filtering

#### 3.1 Methodology

The following procedure is used to reduce the set of indirect targets based on type considerations. It constructs a sound over-approximation of the program’s call graph which we call Complete Call Graph (CCG).

We start with an empty CCG. First, the direct call graph (DCG) rooted at the program’s entry point is added to the CCG. Functions called during initialization and termination and their corresponding DCGs are also added to the CCG. This includes library constructors, .preinit_array, .init_array, .fini_array, .init, .fini. Next, we select from the list of functions identified as potential indirect targets by Sysfilter, those whose type matches a function pointer found in the CCG. All the selected functions and their direct call graph get added to the CCG. The process is repeated recursively as adding those functions might introduce some new function pointer types and therefore some new potential indirect targets.

More precisely, the current implementation proceeds as follows:

1. Run Sysfilter to obtain the initial list of indirect targets (the goal is to prune this list).
2. Determine every address-taken function’s type.
3. Add every function of the DCG rooted at the program’s entry point to the CCG.
4. Add every function of the DCG rooted at functions executed on initialization/termination to the CCG.
5. Find the types of the function pointers that are in scope in the CCG.
6. Add address-taken functions whose type matches any function pointer type found in the previous step and that do not already appear in the CCG.
7. If any function was added, go to step 5.

#### 3.2 Finding address-taken function types

The type of a function (return type and arguments types) can be obtained from DWARF debug symbols. Each function is represented in the debugging information by an entry with the tag DW_TAG_subprogram. The DW_AT_name attribute of this entry is the function’s name and the DW_AT_type attribute represents the function’s return type. The child entries that have a DW_TAG_formal_parameter tag represent the function’s arguments and their DW_AT_type attribute corresponds to their respective types. By reading the debug symbols as shown in Section 2, we determine the type of every indirect target found by Sysfilter.

#### 3.3 Finding potential indirect call types

Determining the type of every function pointer in scope gives a criteria to filter potential indirect targets. This is based on the observation that the address of an indirect target would
typically get stored in a function pointer of matching type. To do so, we look for function pointers in variables (local and global), arguments, and function return values. Specifically, in step 5 of the procedure described in Section 3.1, we iterate through all the functions in the CCG and look for function pointer types in their local variables, arguments and return value then iterate through all the compilation units to find potential function pointer types in global variables.

A more precise approach, that would lead to a tighter overapproximation, would be to obtain the types of the indirect call sites. However, the debugging information doesn’t contain the types of the function called at an indirect call site. DWARF 5 [2] introduced the DW_TAG_call_site DIE and according to the standard, in the case of indirect calls, its DW_AT_call_origin entry may point to the entry describing the function pointer and it could even have DW_AT_type attribute pointing to an entry describing the type of the called function. Unfortunately, current compilers (this was verified on GCC v12.1.0 and Clang v14.0.3-2) emit DW_TAG_call_site with neither DW_AT_call_origin or DW_AT_type attributes for indirect call sites. This makes it impossible to determine the type of the function called.

The DW_TAG_call_site has DW_TAG_call_site_parameter child entries which represents the parameter passed to the function. Their attributes usually only describe the registers used to pass the parameters but not their type. This could be used to filter out some indirect targets for a given call site by considering the number of arguments. Indeed we know that the functions called at a call site have at least as many arguments as the number of DW_TAG_call_site_parameter entries at that call site. But this condition is not restrictive enough and doesn’t lead to any reduction in practice, especially since an address taken function should not be a potential target of any single call site in order to be removed. In the future, if compilers emit the DW_TAG_call_site and its DW_AT_call_origin attribute as described in Section 3.4.1 of the DWARF 5 standard [2], this would provide an exact list of the function pointer types called in a program and allow for significant filtering of indirect targets.

4 Temporal Specialization

Temporal system call specialization aims to determine the sets of system calls needed by a program at different stages of its execution. For example, it is common for an application to only use certain syscalls during initialization and it would be interesting to restrict their use once they are no longer needed.

Existing work [4] requires developers to manually annotate execution phases. In contrast, we attempt to automatically find different stages of the program and their corresponding syscalls sets. To do so we consider thread entry points and function which call fork as they often semantically corresponds to self-contained execution phases.

With Sysfilter, the analysis can be started at an arbitrary point in the program. However, to get a sound result, every AT-function of the program then has to be added to the call-graph. In most cases this results in no reduction in syscalls or indirect targets when compared with the full program analysis.

Using type filtering allows us to reduce the set of AT-functions further when starting the analysis later in the program’s execution. Indeed, it is straightforward to adapt the methodology presented in Section 3.1 to analyze the program starting from an arbitrary point instead of the program’s main entry point. It suffices to replace the DCG of step 3 with the DCG rooted at the desired entry point. We would then consider fewer function pointers in step 5 and eliminate more functions in step 6.

5 Temporal System Call Policy Enforcement

Sysfilter’s policy enforcement consist in creating a dynamic shared object which, when loaded, installs a seccomp BPF filter and patching the target ELF to automatically load it at launch.

To implement temporal enforcement for threads, we generate one library per thread entry point. When loaded, each library installs the filter corresponding to that entry point. Then, we hook pthread_create to first find the appropriate policy and spawn a thread that will install the new filter before calling the thread’s start routine.

For this proof of concept, only threads whose start routine could be determined statically were considered. In practice it would be possible to generate policies for every potential thread target (every address taken function that matches the pthread_create startRoutine prototype for example) and load the appropriate filter at runtime when the thread is created.

6 Validation

To validate the correctness of the generated filters, we used programs which provide test suites. They were run using no filtering, the policies created by Sysfilter and Sysfilter+ with temporal policy enforcement for threads. In Table 1, “Pass” means that the results of the tests were the same in all three cases.

<table>
<thead>
<tr>
<th>Application</th>
<th>Tests</th>
<th>Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>M4</td>
<td>236</td>
<td>Yes</td>
</tr>
<tr>
<td>MariaDB</td>
<td>2059</td>
<td>Yes</td>
</tr>
<tr>
<td>Nginx</td>
<td>356</td>
<td>Yes</td>
</tr>
<tr>
<td>Redis</td>
<td>81</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLite</td>
<td>31190</td>
<td>Yes</td>
</tr>
<tr>
<td>Vim</td>
<td>255</td>
<td>Yes</td>
</tr>
<tr>
<td>Wget</td>
<td>130</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Summary of the correctness validation test suite.
7 Evaluation

7.1 Type filtering

Table 1 and Table 2 compare the numbers of indirect targets and system calls detected by vanilla Sysfilter with the numbers obtained after adding type filtering (Sysfilter+).

<table>
<thead>
<tr>
<th>Application</th>
<th>Sysfilter</th>
<th>Sysfilter+</th>
<th>Difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>1121</td>
<td>1061</td>
<td>-5.4%</td>
</tr>
<tr>
<td>Git</td>
<td>1826</td>
<td>1706</td>
<td>-6.6%</td>
</tr>
<tr>
<td>Apache</td>
<td>800</td>
<td>743</td>
<td>-7.1%</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>434</td>
<td>372</td>
<td>-14.3%</td>
</tr>
<tr>
<td>Bind</td>
<td>2805</td>
<td>2565</td>
<td>-8.6%</td>
</tr>
<tr>
<td>Redis</td>
<td>1786</td>
<td>1277</td>
<td>-28.5%</td>
</tr>
<tr>
<td>Wget</td>
<td>5815</td>
<td>1520</td>
<td>-73.9%</td>
</tr>
<tr>
<td>Memcached</td>
<td>538</td>
<td>473</td>
<td>-12.1%</td>
</tr>
</tbody>
</table>

Table 2: Number of indirect targets with Sysfilter and Sysfilter+.

<table>
<thead>
<tr>
<th>Application</th>
<th>Sysfilter</th>
<th>Sysfilter+</th>
<th>Difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>120</td>
<td>119</td>
<td>-0.8%</td>
</tr>
<tr>
<td>Git</td>
<td>114</td>
<td>113</td>
<td>-0.9%</td>
</tr>
<tr>
<td>Apache</td>
<td>110</td>
<td>109</td>
<td>-0.9%</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>108</td>
<td>106</td>
<td>-1.9%</td>
</tr>
<tr>
<td>Bind</td>
<td>120</td>
<td>119</td>
<td>-0.8%</td>
</tr>
<tr>
<td>Redis</td>
<td>118</td>
<td>117</td>
<td>-0.8%</td>
</tr>
<tr>
<td>Wget</td>
<td>115</td>
<td>114</td>
<td>-0.9%</td>
</tr>
<tr>
<td>Memcached</td>
<td>109</td>
<td>108</td>
<td>-0.9%</td>
</tr>
</tbody>
</table>

Table 3: Number of system calls with Sysfilter and Sysfilter+.

7.2 Temporal specialization

The temporal specialization numbers in Table 3 and Table 4 where obtained by starting the analysis at a thread entry point or at a function which calls fork. Note that sometimes, multiple entry points are available (if there are multiple threads for example) but usually the number of system calls is the same regardless of the chosen entry point.

In some of these examples, selecting thread entry points and functions which call fork seems to be a good heuristic for finding transition points in the program. For instance, in Nginx, using temporal specialization removes system calls such as clone, execve, listen, getpid and umask which are typically used during the initialization.

In the case of Apache, a function pointer whose type matches main’s type is found. Therefore, main gets added as a potential indirect targets which is why we do not observe any reduction in terms of indirect targets or system calls.

7.3 Large-scale analysis

We performed a large-scale study considering all C/C++ applications in Debian Sid to evaluate the impact of using type information in debug symbols both when starting the analysis from the program’s main entry point and when considering thread entry points and functions which call fork. 22236 programs were analyzed out of which 5088 had threads where the start routine could be resolved statically. When considering thread entry points, Sysfilter+ remove 35% more indirect targets and 4% more system calls than vanilla Sysfilter's analysis of the whole program does.

The analysis time can be seen in Figure 1. The median analysis time is 65 seconds.
Figure 1: Extraction tool runtime per binary (seconds).

Figure 2 and Figure 3 show the reduction in indirect targets and syscalls respectively when comparing vanilla Sysfilter, Sysfilter+ analyzing the whole program and Sysfilter+ starting at thread entry points. When considering the whole binary, using type information reduces the number of indirect targets significantly (33% on average) but it rarely reduces the number of syscalls. On the other hand, when going from main to a thread entry point, the number of indirect targets is only reduced by 2% on average but there is an observable reduction in the number of system calls.

8 Limitations

8.1 Reliability of debug information

The analysis presented here relies on the accuracy of the debug information. However in some cases the debug information may differ from the actual implementation. For example, compilers typically allow the main function to be declared with various prototypes such as the following:

1. `int main(void);`
2. `int main(int argc, char* argv[]);`
3. `int main(int argc, char* argv[], char* envp[]);`

But they all get compiled to:

1. `int main(int argc, char* argv[], char* envp[]);`

Because of this, the type of the function pointer corresponding to the main function’s address getting taken (in _libc_start_main) may differ from type of the main function in the debugging information. The main function would then be erroneously eliminated as no function pointer matches its type. Therefore, main is currently handled as a special case.

If other functions get compiled to a type which is different than what appears in their debugging information, they might get removed by mistake.

8.2 Casts into function pointers

The methodology presented in Section 3 assumes that functions called indirectly will first have their address stored in a variable of matching function type. However if an address is casted into a function pointer and called immediately (without being stored in and intermediate variable), the corresponding
function pointer type will not appear in the debug information and it will be missed.

For example, in the following code, the function pointer type `void *(*)(void *, int)` does not appear in the debug information of the `indirect_call_cast` function.

```c
typedef struct {
    void *(*start_routine)(void *, int);
} ThreadArgs;

static void indirect_call_cast(void* args){
    ((ThreadArgs*) args)->start_routine(NULL, 0);
}
```

References

[1] Detailed results showing the list of syscalls removed for each program. https://docs.google.com/spreadsheets/d/1om0cadmrJVfPo_g8ciiDJ27L1bYK7klY9TjssAMGObk/edit?usp=sharinghere.


Appendix

A Detailed results

The full list of syscalls removed for each program is available online [1].