AutoISES: Automatically Inferring Security Specifications and Detecting Violations

Lin Tan[1]
University of Illinois, Urbana-Champaign
lintan2@cs.uiuc.edu

Xiaolan Zhang
IBM T.J. Watson Research Center
cxzhang@us.ibm.com

Xiao Ma[†‡], Weiwei Xiong[†], Yuanyuan Zhou[†‡]
[†]*University of Illinois, Urbana-Champaign* [‡]*Pattern Insight Inc.*
{*xiaoma2, wxiong2, yyzhou*}*@cs.uiuc.edu*

Abstract

The importance of software security cannot be overstated. In the past, researchers have applied program analysis techniques to automatically detect security vulnerabilities and verify security properties. However, such techniques have limited success in reality because they require manually provided code-level security specifications. Manually writing and generating these code-level security specifications are tedious and error-prone. Additionally, they seldom exist in production software.

In this paper, we propose a novel method and tool, called A*uto*ISES, which Automatically Infers Security Specifications by statically analyzing source code, and then directly use these specifications to automatically detect security violations. Our experiments with the Linux kernel and Xen demonstrated the effectiveness of this approach – AutoISES automatically generated 84 security specifications and detected 8 vulnerabilities in the Linux kernel and Xen, 7 of which have already been confirmed by the corresponding developers.

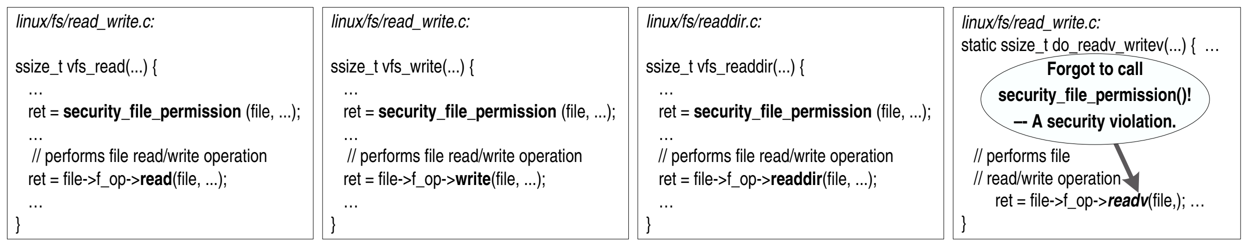
1 Introduction

1.1 Motivation

The critical importance of software security has driven the design and implementation of secure software systems. Security-Enhanced Linux (SELinux) [23, 28], developed as a research prototype to incorporate Mandatory Access Control (MAC) into the Linux kernel several years ago, imposes constraints on its existing Discretionary Access Control (DAC) for stronger security. SELinux has since been adopted by the mainline Linux 2.6 series and incorporated into many commercial distributions, including Redhat, Fedora, and Ubuntu. Recently, Xen also adopted a similar MAC security architecture to enable system-wide security policy [7].

A core part of such access control systems is a set of security check functions, which check whether a subject (e.g., a process) can perform a certain operation (e.g., read or write) on an object (e.g., a file, an inode, or a socket). These protected operations are called *security sensitive operations*. For example, Linux's security check function `security_file_permission(`⟨*file*⟩`, ...)` can determine if the current process is authorized to read or write the file, while another security check function, `security_file_mmap(`⟨*file*⟩`, ...)`, checks if the current process is authorized to map a file into memory. To ensure only authorized users can read or write the file, developers must add the security check function `security_file_permission()` before each file read/write operation on every file. Similarly, developers must add `security_file_mmap()` each time before mapping a file to memory, to ensure only authorized users can memory map the file.

A major challenge of supporting the secure architecture above is to ensure that all sensitive operations on all objects are protected (i.e., checked for authorization) by the proper security check functions in a consistent manner. If the proper security check function is missing before a sensitive operation, an attacker with insufficient privilege will be able to perform the security sensitive operation, causing damage. For example, the file read/write operation is performed in many functions throughout the Linux kernel, such as `read()`, `write()`, `readv()`, `writev()`, `readdir()`, and `sendfile()`. Despite the different names of these function calls, they all perform the same *conceptual* file read/write operation, and must be checked for authorization by calling the security check function `security_file_permission()`. As the Linux kernel code is reasonably mature, most of these functions performing the file read/write operation, such as `read()`, `write()`, and `readdir()`, are protected by



```
linux/fs/read_write.c:

ssize_t vfs_read(...) {
  ...
  ret = security_file_permission (file, ...);
  ...
   // performs file read/write operation
  ret = file->f_op->read(file, ...);
  ...
}
```

```
linux/fs/read_write.c:

ssize_t vfs_write(...) {
  ...
  ret = security_file_permission (file, ...);
  ...
   // performs file read/write operation
  ret = file->f_op->write(file, ...);
  ...
}
```

```
linux/fs/readdir.c:

ssize_t vfs_readdir(...) {
  ...
  ret = security_file_permission (file, ...);
  ...
  // performs file read/write operation
  ret = file->f_op->readdir(file, ...);
  ...
}
```

```
linux/fs/read_write.c:
static ssize_t do_readv_writev(...) { ...

  // performs file
  // read/write operation
      ret = file->f_op->readv(file,); ...
}
```

Forgot to call security_file_permission()! — A security violation.

(a) file read/write operation protected by the check

(b) file read/write operation protected by the check

(c) file read/write operation protected by the check

(d) **Security Violation –** violating the implicit security specification

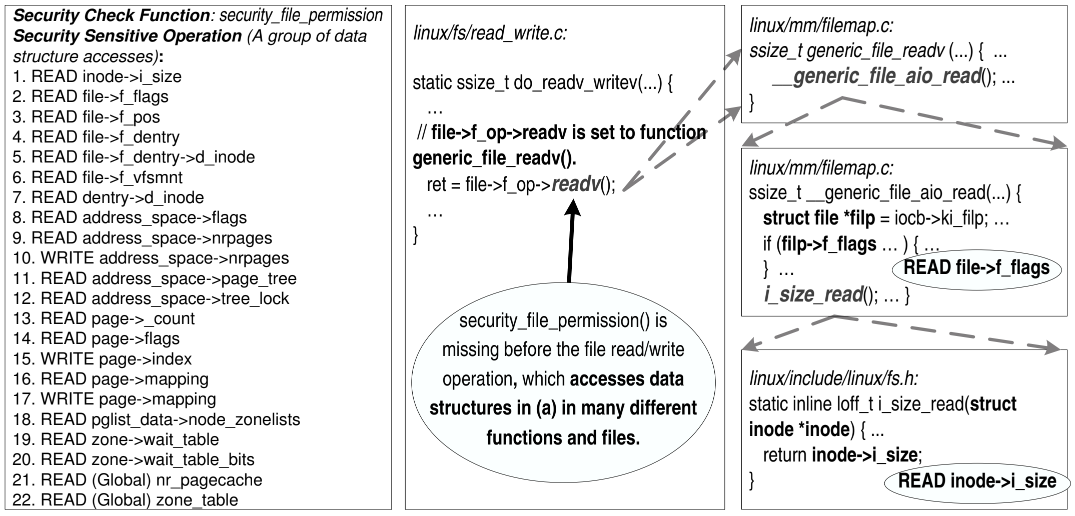
Figure 1: A real security violation in Linux 2.6.11. The security check `security_file_permission()` was missing before the security sensitive operation performed via `file->f_op->readv()`, violating the implicit security specification — every file read/write operation must be checked for authorization using `security_file_permission()`. This is a real security violation, which has already been fixed in later versions. The code is slightly modified to simplify illustration.

the security check function, as shown in Figure 1(a)-(c). However, in a few other cases, as shown in Figure 1(d), the security check function is not invoked before the file read/write operation performed by `readv()`, violating the *implicit* security specification or security rule: every file read/write operation must be protected by calling security check function `security_file_permission()`. Due to this real world security vulnerability in Linux 2.6.11 (*CVE-2006-1856* [1]), unauthorized user can read and write files that they are not allowed to access, potentially providing unauthorized user account access. Additional damages might include partial confidentiality, integrity, and availability violation, unauthorized disclosure of information, and disruption of service.

There have been great advances in applying program analysis techniques [2, 4, 5, 12, 16] to automatically detect these security vulnerabilities and to verify security properties [6, 9, 18, 30]. Generally, these tools take a specification that describes the security properties to verify as input. For example, in earlier efforts [9, 30], the authors *manually* identified the data types (e.g., `struct file, struct inode`, etc.) that might be accessed to perform security sensitive operations and automatically verified that any access to these data types was protected by a security check function. Although these previous studies detected some vulnerabilities, and made significant progresses toward automatic verification of security properties, they are limited in two perspectives:

- All these previous tools require developers or their tool users to provide code-level security specifications, which greatly limit their practicability in checking and verifying security properties. Writing specifications that accurately capture the security properties of a piece of software and at the same time maintaining their correctness across different versions of the software is notoriously difficult. Such specifications seldom exist in production software.
- Human-generated specifications can be imprecise, causing false positives and potentially false negatives in violation detection. As an example, the specification used in one of the earlier work [30], introduced false positives because it treated *any* access to specified data structures as security sensitive operations. In reality, a security sensitive operation typically consists of accesses to multiple data structures: a file read/write operation involves accessing `struct file`, `struct inode`, `struct dentry`, etc. Accessing the file structure alone is not necessarily (actually in most cases is not) a file read/write operation. In addition, some field accesses (e.g., `file->f_version`) of a security sensitive data type are not part of any security sensitive operation, and therefore do not need to be protected. Therefore, in the two cases above, simply requiring accesses to every field of these data structures to be protected led to false positives [30]. The specification may also introduce false negatives because it does not specify which security check is required for which operation. The tool can fail to detect violations where the wrong check function is used as different security sensitive operations (e.g., file read/write and memory map) may access the same data types (e.g., `struct file`) but require different security checks.

Therefore, to design tools that are truly usable for ordinary programmers, it is highly desirable for these tools to meet the following three requirements on specification generation: (1) to automatically check against source code for security violations, the security specifications must be at the code level. The conceptual specification "file read/write operation must be protected by the security check function `security_file_permission()`" can not be checked against source code without knowing its corresponding code-level representation. (2) as it is tedious and error-prone for developers to write these



Security Check Function: security_file_permission
Security Sensitive Operation (A group of data structure accesses)**:**

1. READ inode->i_size
2. READ file->f_flags
3. READ file->f_pos
4. READ file->f_dentry
5. READ file->f_dentry->d_inode
6. READ file->f_vfsmnt
7. READ dentry->d_inode
8. READ address_space->flags
9. READ address_space->nrpages
10. WRITE address_space->nrpages
11. READ address_space->page_tree
12. READ address_space->tree_lock
13. READ page->_count
14. READ page->flags
15. WRITE page->index
16. READ page->mapping
17. WRITE page->mapping
18. READ pglist_data->node_zonelists
19. READ zone->wait_table
20. READ zone->wait_table_bits
21. READ (Global) nr_pagecache
22. READ (Global) zone_table

```
linux/fs/read_write.c:

static ssize_t do_readv_writev(...) {
  ...
 // file->f_op->readv is set to function
generic_file_readv().
  ret = file->f_op->readv();
  ...
}
```

security_file_permission() is missing before the file read/write operation, which accesses data structures in (a) in many different functions and files.

```
linux/mm/filemap.c:
ssize_t generic_file_readv (...) {  ...
     __generic_file_aio_read(); ...
}
```

```
linux/mm/filemap.c:
ssize_t __generic_file_aio_read(...) {
  struct file *filp = iocb->ki_filp; ...
  if (filp->f_flags ... ) { ...
  }  ...
  i_size_read(); ... }
```

```
linux/include/linux/fs.h:
static inline loff_t i_size_read(struct
inode *inode) { ...
  return inode->i_size;
}
```

(a) Security Rule/Specification (b) Security Violation (c) Accesses in the Code

Figure 2: A code-level security specification AutoISES automatically generated and a real security violation to the specification in Linux 2.6.11. The leftmost box shows the security rule, consisting of a security check function and a group of data structure accesses. Each row is one access, which can be either a structure field access or a global variable access, denoted by "(Global)". For each structure field access, the name before the first −> is the type name of the structure, and the rest are field names. For a global variable, the variable name is used. The code is slightly modified to simplify illustration.

security rules, the tool should automatically generate security specifications with minimum user/developer involvement; and (3) the generated specification must be precise, otherwise it would result in too many false positives and/or false negatives.

1.2 Our Contributions

This paper makes two contributions:

(1) We propose an approach and a tool, AutoISES, to automatically extract concrete code-level security specifications from source code and automatically detect security violations to these specifications. Our key observation is that although the same security sensitive operation can be performed in different functions, ultimately, the structure fields and global variables these functions access are the same. We call these structure field and global variable accesses together as *data structure accesses*. For example, all of the different functions performing the file read/write operation share the 22 data structure accesses listed in Figure 2(a) (automatically generated by AutoISES), including reading field `f_flags` of the `file` structure and reading field `i_size` of the `inode` structure. These 22 data structure accesses are performed in many different functions located in different source files. Intuitively, this makes sense as security check functions are designed to protect *data*. Therefore, the use of data structure accesses is *fundamental* in representing security sensitive operations.

Based on this observation, we propose a method and a tool, called A*uto*ISES, to Automatically Infer Security Specifications by statically analyzing source code and *directly* use these specifications to automatically detect security violations. Specifically, if a code-level security sensitive operation is frequently protected by a security check function in source code, AutoISES automatically infers that the security check function should be used to protect the particular code-level security sensitive operation. Our *rationale* is that for release software, the majority of the code should be correct, therefore we can use the code to infer security specifications or rules, which are observed in most places of the source code, but may not in a few other places. The rationale is similar to that of prior work in specification mining [10, 11, 20, 22], each of which extracts different types of programming rules automatically from source code or execution trace. However, *previous techniques are not directly applicable to our problem*, because they are limited by the types of rules they can infer (e.g., function correlation rules [10, 20], variable value related invariants [11], variable pairing rules [22]). As noted previously, our key observation states that the sensitive operation should be represented as data structure accesses, therefore, it requires AutoISES to be able to learn specifications that contain both functions and multiple variable accesses that satisfy certain constraints. None of the previous techniques can be applied without significant re-design of the learning algorithm (more detailed discussion in Section 6.1 and Section 7).

We evaluated AutoISES on the latest versions of two large software systems, the Linux kernel and Xen, to demonstrate the effectiveness of our approach. AutoISES automatically extracted 84 rules from the Linux kernel and Xen, and detected 8 true violations, 7 of which are confirmed and fixed by the corresponding developers. Figure 2 shows (a) the code-level security specification learned by AutoISES which consists of the 22 data structure accesses, (b) a security violation automatically detected by AutoISES, and (c) the unprotected sensitive operation that performs all the accesses shown in (a) in different functions located in various source files. It would be very difficult, if impossible, for a human being to generate such a specification. More examples and results can be found in Section 5.

The automatically generated specifications can also be used by other analysis tools for vulnerability detection. Additionally they can assist in software understanding and maintenance. These results demonstrate that AutoISES is effective at automatically inferring security rules and detecting violations to these rules, which greatly improves the practicality of security property checking and verification tools.

(2) We quantitatively evaluate rule granularity impact on the accuracy of rule inference and violation detection. Security specifications can vary in *granularity*. For example, a single access can be represented with the access type (read or write), `READ inode->i_size`, or without it, `ACCESS inode->i_size`. Similarly, the same access can be represented with the structure field, `READ inode->i_size`, or without it, `READ inode`. Theoretically, finer granularity causes more false negatives and fewer false positives for violation detection compared to coarser granularity. The choice of granularity can greatly affect the accuracy of rule inference and violation detection. Almost all previous rule generation and violation detection techniques [3, 9, 10, 11, 20, 22, 30] choose fixed granularity without *quantitatively* evaluating how good their choice is.

In our work, we quantitatively evaluate the impact of different rule granularity on rule inference and violation detection. This approach is *orthogonal* to our automatic rule inference techniques and can be applied to other rule extraction techniques.

Interestingly, our results show that if we do not distinguish the fields, then the inferred code-level security sensitive operation for the check function `security_inode_link()` and `security_file_unlink()` is the same, failing to distinguish the two different operations. Using our finest granularity, AutoISES can disambiguate the two similar operations (the unlink operation contains `READ inode->i_size`, but the link operation does not). Our results also show that on average our most fine-grained rules causes 33% fewer false positives than the most coarse-grained rules, and detected all of the true violations that the most coarse-grained rules can detect. This indicates rule granularity significantly affects violation detection accuracy and could be considered a tuning parameter for other rule inference and violation detection tools to reduce false positives.

On the other hand, coarse-grained rules help us discover high level rules that are shared by different security check functions, which fine-grained rules fail to uncover (examples shown in Section 5.3). These results call for attention that different levels of granularity have measurable advantages and disadvantages, and one could quantitatively evaluate the tradeoffs when designing rule inference and violation detection tools in order to choose the most suitable granularity.

In summary, AutoISES closes an important gap in achieving secure software systems. To have truly secure software systems, not only must one have a secure design, but the implementation must faithfully realize the design. To verify that the implementation faithfully realizes the design, one must write a correct code-level specification which can be verified by automatic tools such as a model checker or a static analyzer. AutoISES allows the security specifications to be automatically extracted from the actual implementation, alleviating the developers from the burden of manually writing specifications while at the same time significantly improving the accuracy of the specification.

1.3 Paper Layout

The remainder of the paper is organized as follows. Section 2 provides background information about MAC, DAC and the assumptions we make in our work. In Section 3, we present an overview of our approach, including some formal definitions and how we quantitatively evaluate rule granularity, followed by a detailed design in Section 4. Our methodology and experimental results are described in Section 5, and Section 6 discusses and summarizes our key techniques, their generalization and limitations. In Section 7 a discussion of the related work is presented, and finally we conclude with Section 8.

2 Background and Assumptions

2.1 DAC and MAC Background

The traditional Linux kernel uses Discretionary Access Control (DAC), meaning the access control policies are set at the discretion of the owner of the objects. For example, the root user typically sets the password file to be writable only by herself. However, if the root user mistakenly makes the password file publicly writable, then the whole system is at risk. This example shows one major deficiency of DAC, that is, mistakes of *individual policy decisions* can result in the breach of security for the *entire* system. Mandatory Access Control (MAC) is proposed to address this issue. In a MAC system, there exists a system wide security policy, such as "high-integrity file must not be modified by low-integrity users". Even if the root mistakenly grants write permissions on the password file to everyone, when a normal user tries to write the password file the attempt would fail because it is against the system-wide policy. MAC is considerably "safer" than DAC, but it is also more complex and more difficult to implement, especially for large systems like Linux [18]. It took Linux developers about two years to add MAC to the Linux kernel, and since then it has undergone many rounds of refinements and extensions. It is expected that its development will continue well into the future.

2.2 Assumptions

We make the following assumptions in our work.

Reasonably mature code base: Similar to previous work on automatic rule extraction [3, 10, 11, 20, 22], we assume that the code we work with is reasonably mature, i.e., it is mostly correct and already contains an implementation of the security architecture that is mostly working. This does not mean that software development ceases. In fact, the software might still be under active development and new features continue to be added. Almost all open source and proprietary software falls into this category, therefore this assumption does not significantly limit the applicability of this work.

Software developers not adversarial: We assume that software developers are trusted and will not deliberately write code to defeat our rule generation mechanism. This in general holds for majority of the software that exists today, i.e., we believe that the majority of software developers intend to write correct and secure software. In the limited cases where this assumption does not hold [25], there exist static analysis techniques that can detect such malicious code [29]. However, detecting malicious code in general is challenging and remains an active open research problem.

Kernel and hypervisor in the trusted computing base: For the two pieces of software that we experimented with, namely, the Linux kernel and the Xen hypervisor (virtual machine monitor), we assume that both are part of the trusted computing base. Thus, the mandatory access control is in place to prevent *user level or guest OS level* processes from breaking security policies. This assumption implies that only if a user process or a guest OS process can bypass the MAC mechanism (placed in the kernel or the hypervisor) do we consider it a breach of security. The kernel or the hypervisor is free to modify the data structures on its own behalf (e.g., for bookkeeping purposes) without going through the security checks. This assumption is adopted from the MAC architecture of the Linux kernel and Xen hypervisor.

3 Overview of Our Approach

In this section, we will present our high level design choices of rule inference and violation detection; formal definitions of security rules, security sensitive operations, and security violations; and how we explore different levels of rule granularity. The detailed design will be discussed in the next section.

3.1 Our approach

Our approach consists of two steps, as shown in Figure 3. In the first step, we generate security specifications automatically from the source code. The input to the generator is the source code and the set of security check functions. The output of this step is a set of security rules containing a security check function and a security sensitive operation represented by a group of data structure accesses as shown in Figure 2 (a) (the advantages of using a group of data structure accesses to represent a sensitive operation are discussed in Section 3.2). In the second step, AutoISES takes the source code and the rules automatically inferred in the first step as the input, and outputs ranked security violations. Note that these automatically inferred rules can be used *directly* by AutoISES without manual examination, which reduces human involvement to its minimum.

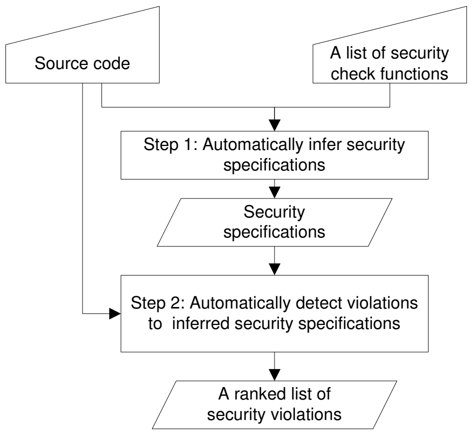

Source code
A list of security check functions
Step 1: Automatically infer security specifications
Security specifications
Step 2: Automatically detect violations to inferred security specifications
A ranked list of security violations

Figure 3: The analysis flow of AutoISES.

Step 1: Inferring Security Rules This paper focuses on inferring *security rules* which mandate that a security sensitive operation must be protected by a security check function, i.e., the sensitive operation must not be allowed to proceed if the security check fails. In order to effectively check such rules against the source code to detect violations, it is crucial to specify the security rule at the source code level. Unfortunately, in reality such rules are usually not documented. Therefore, our goal is to automatically infer such rules from the source code.

To infer one rule, we want to discover, for the *same* security check function, what *fixed* security sensitive operation must be protected by it. We can infer this security rule from two angles: (1) we search for all instances of *the same security check function* (e.g., `security_file_permission()`) and discover what sensitive operation is frequently protected by (e.g., performed after) the check function, or (2) we search for all instances of *the same security sensitive operation* (e.g., the 22 data structure accesses shown in Figure 2(a)) and then check what security check function is frequently used to protect (e.g., invoked before) the operation. We use the first method, because it is relatively easy to know what the security check functions are in the source code (usually documented), but knowing what security sensitive operations are in the source code itself is still a challenge (not documented). Specifically, we look for all instances of the same security check in the source code and collect sensitive operations protected by it. If this security check is frequently used to protect a fixed sensitive operation, represented by a fixed set of data structure accesses, we infer a security rule: this set of data structure accesses must be protected by this security check function. Our rationale is that released software is mostly correct, so we can infer correct behavior from it.

It is not uncommon that more than one security check function is required to protect one sensitive operation. In such cases, our inference approach still works because it will infer several separate rules, one for each security check function. The set of rules related to the same sensitive operation combined can detect violations where not all of the check functions are invoked to protect the operation.

We can infer security rules statically or dynamically. While a dynamic approach is more precise, it has poorer coverage because only executed code is analyzed. As we study large software with millions lines of code, a dynamic approach may not be sufficient, which is confirmed by previous work [9, 13]. Therefore, we use *inter-procedural* and *flow-insensitive* static program analysis for rule inference. A more detailed description of our static analysis techniques can be found in Section 4.3.

In summary, our tool AutoISES automatically infers sensitive operations in the form of a group of data structure accesses that are commonly or frequently protected by the same security check function, given a list of security check functions. Similar to previous rule inference studies [3, 10, 11, 20, 22], we cannot discover all security rules from the source code alone (discussed in Section 6.3). However, it is effective to infer some important security rules from source code, and detect previously unknown security vulnerabilities.

Step 2: Detecting Violations The goal of this step is to use the rules inferred above to detect security violations. Similarly, we use inter-procedural and flow-insensitive analysis for violation detection. As we already know which data structure accesses represent the security sensitive operation from an inferred rule, we can search for instances of the security sensitive operation that are not protected by the security check function, indicating security violations.

Specifically, AutoISES starts from each root function (*automatically* generated starting function for our analysis and detection as discussed later in Section 4.2.1), and collects all data structure accesses and calls to security check functions. Then it calculates the $accessViolationCount$, which is the number of accesses in the rule that are not protected by the particular security check function. Specifically, if an access in the rule is performed without being protected by a security check function, AutoISES increases the $accessViolationCount$ by one. We then use the $accessViolationCount$ for violation ranking – the higher the $accessViolationCount$ is, the more likely it is a true violation. We also allow our tool users to set up a threshold and only report violations with its $accessViolationCount$ higher than the threshold. Users can always set the threshold to zero to see all violation reports.

Untrusted-space exposability analysis One key technique we used to greatly reduce false positives is our *untrusted-space exposability* analysis. As we consider the kernel and the hypervisor to be our trusted computing base, security sensitive operations in kernel space and hypervisor that do not interact with the untrusted space (user space or guest OS processes), do not need to be protected by a security check function. On the other hand, if such sensitive operations interact with the untrusted space, e.g., are performed by a user space process via system calls, or use data copied from user space, then a security check may be mandatory. Since it is typical that a large number of sensitive operations are not exposed to the untrusted space, most of the detected violations would be false alarms, which is detrimental to a detection tool.

To reduce such false positives, we perform a simple trusted space exposability study. Specifically, we compiled a list of user space interface functions that are known a priori to be exposed to user space, e.g., system calls such as `sys_read()` and hypercalls. Then, AutoISES checks what sensitive operations are reachable from these interface functions. If a sensitive operation that can be exposed to the untrusted space is not protected by the proper security check function, we report the violation as *an error*; otherwise, we report the violation as *a warning*. Our goal is to ensure most of the errors are true violations, but we still generate the warnings so that developers can examine them if they want to. This approach relies on easy-to-obtain information (system calls and hypercalls) to automatically reduce the number of false positives.

3.2 Formal Definitions

Based on our reasoning above, we formally define the rule inferencing problem, security sensitive operations, security rules, our inference rule, and violations.

Rule Inferencing Problem Given the target source code and a set of n kernel *security check functions*, $CheckSet = \{Check_1, ..., Check_n\}$, each of which can check if a subject (e.g., a process), is authorized to perform a certain *security sensitive operation*, Op_i (e.g., read, where $1 \leq i \leq n$), on a certain object (e.g., a file) we want to uncover *security specifications* or *security rules*, $Rule_i$, in the form of a pair, ($Check_i$, Op_i), mandating that a security sensitive operation Op_i, must be *protected*, $<_{protected}$, by security check function $Check_i$ each time Op_i is performed. Here protected means that the operation Op_i can not be performed if the check $Check_i$ fails.

A security check function $Check_i$ can be called multiple times in the program, each of which is called an instance of the security check function, denoted as $InstanceOf(Check_i)_v$, where v is between 1 and the total number of $Check_i$ instances inclusive. Similarly, a security sensitive operation Op_i can appear in the program multiples times, and each of which is called an instance of the sensitive operation, $InstanceOf(Op_i)_u$. If for all instances of the sensitive operation, there exists at least one instance of security check function to protect it, then we say that the sensitive operation is protected by the security check function. Formally speaking, $\forall InstanceOf(Op_i)_u, \exists InstanceOf(Check_i)_v$, such that $InstanceOf(Op_i)_u <_{protected} InstanceOf(Check_i)_v => Op_i <_{protected} Check_i$.

Representing Security Sensitive Operations There are several ways to represent security sensitive operations at the code level. We can use a list of data structures that the operation manipulates, a list of functions the operation invokes, or the combination of the two. The list can be ordered or not ordered, indicating whether we require these accesses to be performed in any particular order.

We use data structure accesses to represent a security sensitive operation, because it has two advantages over using function calls. First, it can infer rules that function call based analysis would not be able to find. For example, if a sensitive operation is performed after a check function via different function calls, e.g., A and B, by using function A and B to represent the operation, we may mistakenly consider nothing is commonly protected by the check function and miss the rule. Zooming into the functions will allow us to find the shared data structure accesses in both A and B. Additionally, we can detect more violations by using data structure accesses. For examples, if we find that a check function always protects function call A at many places, but there is a violation that performs the same sensitive operation via function B without invoking the check function first, then we will not be able to detect the violation unless we use the data structure accesses to represent the sensitive operation in the rule. For example, `security_file_permission()` is used to protect `read`, `write`, etc., in Linux 2.6.11, but the check is missed when the sensitive operation is called through `readv` (shown in Figure 2(b)), `writev`, `aio_read`, or `aio_write`. Therefore, AutoISES would have missed all of these violations if it had not used the actual data structure accesses to represent the sensitive operation.

The tradeoffs between considering access orders or not are as follows. While preserving access orders is more precise, it has two major disadvantages. First, the order does not matter for certain rules, and preserving the order can cause one to miss the rule. For example, an directory removal operation involves setting the inode's size to 0 and decrement the number of links to it by one. The order in which the two accesses are performed is irrelevant. Second, it is more expensive to consider access orders, which can affect the scalability of our tool. On the other hand, the downside of not considering orders is that we can potentially have a higher number of false positives due to over-generalization. However, we did not find any false positives caused by this reason in this study.

Therefore, we use a set of unordered data structure accesses, $AccessSet = \{Access_1, ..., Access_m\}$, to represent sensitive operation Op, where each data structure access is defined as shown in Figure 4.

$$
\boxed{\begin{array}{l} Access_i := READ\ AST \mid WRITE\ AST \mid ACCESS\ AST \\ AST := type\_name(->field)* \mid global\ variable \end{array}}
$$

Figure 4: Definition of one data structure access. $ACCESS\ AST$ means an access to AST (Abstract Syntax Tree), either READ or WRITE.

Security Rules Replacing the security sensitive operation Op_i with $AccessSet$ as defined above, we have the following definition of security rules:

$$
Rule_i = (Check_i, AccessSet_i),\ where\ Check_i \in CheckSet
$$
$$
=> AccessSet_i <_{protected} Check_i.
$$

Inference Rule As such rules are usually undocumented, we want to automatically infer them from source code by observing what sensitive operation is frequently protected by a security check function, i.e., what sensitive operation are commonly protected by different instances of the same security check function.

Formally speaking, we use the following inference rule to infer security rules:

$$AccessSet_i <_{frequently\ protected}\ Check_i$$
$$=>\ InferredRule_i = (Check_i, AccessSet_i),$$
$$where\ Check_i \in CheckSet.$$

Violations Using such inferred rules, we want to detect security violations. An instance of a security sensitive operation, $InstanceOf(AccessSet_i)_u$ is a violation to $InferredRule_i$ if it is not protected by any instance of the security check function. In other words,

$$Given\ InferredRule_i = (Check_i, AccessSet_i),$$
$$\forall\ InstanceOf(Check_i)_v,$$
$$InstanceOf(AccessSet_i)_u \not<_{protected}\ InstanceOf(Check_i)_v$$
$$=>\ InstanceOf(AccessSet_i)_u \in Violation_i.$$

In this paper, we use rules and inferred rules interchangeably.

3.3 Exploring Rule Granularity

We explore 4 different levels of granularity based on two metrics, whether to distinguish read and write access types, and whether to distinguish structure fields. The four different levels of granularity are as shown in Table 1. For example, the access `READ inode->i_size` is represented as `READ inode` for Granularity($F-$, $A+$), `ACCESS inode->i_size` for Granularity($F+$, $A-$), and `ACCESS inode` for Granularity($F-$, $A-$).

		Distinguishing Structure Fields	
		Yes	No
Distinguishing Access Types	Yes	Granularity($F+$, $A+$) READ inode->i_size	Granularity($F-$, $A+$) READ inode
	No	Granularity($F+$, $A-$) ACCESS inode->i_size	Granularity($F-$, $A-$) ACCESS inode

Table 1: Four Levels of Rule Granularity with Examples.

To better understand the impact of the rule granularity on rule inference and violation detection, and to gain insight on how well our default granularity (Granularity($F+$, $A+$)) performs, we quantitatively evaluate the 4 different levels of granularity on the Linux kernel and Xen. This exploration is orthogonal to our rule inference and violation detection, and can be applied to previous rule inference techniques [9, 11, 20, 22, 30].

4 Detailed Design of AutoISES

4.1 A Naive Approach

We first describe a naive approach and show why it does not work, which motivated us to explore alternatives. A naive approach is to start the analysis from the direct caller functions of a security check function, and consider all data structure accesses performed after the security check function as the protected sensitive operation. This approach does not work because it introduces obvious imprecision. For example, as shown in Figure 5, `security_inode_permission()` is called at the end of function `permission()`. If we start from function `permission()`, then no data structures are accessed after security check function `security_inode_permission()` in function `permission()`, indicating that no data structure access is protected by `security_inode_permission()`, which is clearly not true. This naive approach fails because `permission()` is not a function that actually uses the check to protect security sensitive operations. Instead, it is a wrapper function of the security check function. The function that actually uses the security check function for a permission check is `vfs_link()` shown in the leftmost box of Figure 5.

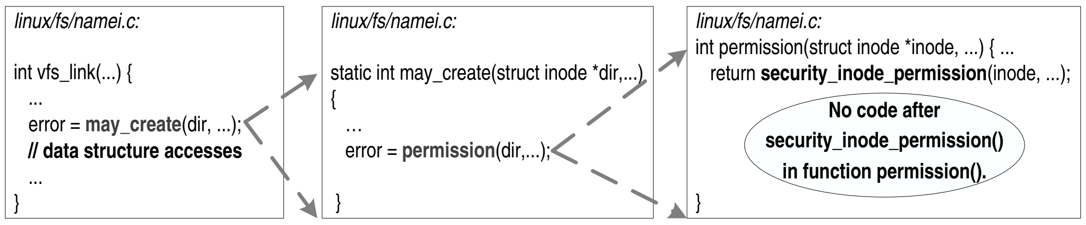
To automatically infer security rules, we need to automatically discover the functions (e.g., `vfs_link()`) that actually use security checks for authorization checking.

4.2 Security Specification Extraction

The goal of AutoISES is to discover the security sensitive operation, represented by a group of data structure accesses, that is protected by a security check function. Why we use data structure accesses to represent a security sensitive operation has already been discussed in Section 1.2, and the two major advantages of this representation have been described in Section 3.2. To achieve this goal, we need to address four major challenges: (1) how to automatically discover functions that actually use security checks for authorization checking; (2) how to define "protected" at the code level; (3) what information to extract; (4) how do we turn such information into security rules.

4.2.1 How to find functions that actually use security checks for authorization checking?

As shown above, simply starting the analysis from the direct callers of a security check function does not work. To automatically detect security rules, we need to automatically find the functions that actually use the check function to protect sensitive operations. However, what functions actually use the check function for authorization checking depends on the semantics of the software, and thus are extremely difficult to extract automatically.



```
linux/fs/namei.c:

int vfs_link(...) {
  ...
  error = may_create(dir, ...);
  // data structure accesses
  ...
}
```

```
linux/fs/namei.c:

static int may_create(struct inode *dir,...)
{
  ...
  error = permission(dir,...);

}
```

```
linux/fs/namei.c:
int permission(struct inode *inode, ...) { ...
  return security_inode_permission(inode, ...);

}
```

Figure 5: Demonstrating the naive approach does not work.

Instead, we try to automatically extract a good approximation of these functions. Specifically, we (1) automatically break the program into modules (e.g., each file system is a module) based on the compilation configurations that come with any software (e.g. in Makefile), and (2) consider the *root functions* of each module as functions that actually use security check functions for authorization check, where *root functions* are functions that are not called by any other functions in the module. These **root functions can be automatically extracted by analyzing the call graphs** of each module.

Using this approach, AutoISES finds that `sys_link()` is a root function for the `ext2` file system module. Although `vfs_link()` is the direct user of the check, this approximation is good because the root function `sys_link()` is the caller of `vfs_link()`, therefore the root function contains all the data structure accesses `vfs_link()` performs. While it can also contain accesses that are not in `vfs_link()`, which may not be related to the security sensitive operation, it does not affect the violation detection accuracy much in practice mainly for two reasons. First, since only accesses that are protected by *many* instances of the same check function is considered as part of a sensitive operation, many unrelated accesses can be automatically eliminated during the rule generation stage (Section 4.2.4). Additionally, during the violation detection stage, we can set the threshold for $accessViolationCount$ lower to tolerate a few unrelated data structure accesses. Note that these root functions are usually a super set of our untrusted space interface functions, as many root functions can only be called by other *kernel* modules, which are considered trusted. Therefore, our untrusted-space exposability study is necessary for reducing false positives.

An alternative solution is to ask developers or tool users to provide the functions that actually use the check functions. Although it is easier to provide such functions than writing the specifications directly, it is not desirable, because (1) it is not automatic; (2) one would need to manually identify such functions each time new code is added; and (3) manually identified these functions can be error-prone.

4.2.2 What does "protected" mean at the code level?

An instance of a sensitive operation Op_i is considered protected by an instance of a security check function $Check_i$, if the operation is allowed only if it is authorized as indicated by the return value of the check function. To implement this exact semantic, we need to know the semantics of return values of all the security check functions, which requires significant manual work and does not scale; this is not desirable. Therefore, we use a source code level approximation of this semantic: a security check function protects all data structure accesses that appear "after" the security check function in an execution trace. Although this approximation can include some unrelated accesses, it is reasonably accurate and effective at helping detecting violations for the same two reasons discussed in Section 4.2.1. Additionally, the approximation makes our approach more automatic and general, because we do not require developers to provide the semantics of the return values of the security check functions.Similar to previous static analysis techniques, our static analysis does not employ any dynamic execution information. Instead, the execution trace we use is a static approximation of the dynamic execution trace.

4.2.3 What information to extract?

We want to extract data structure accesses that are frequently protected by a security check function. Since a typical program accesses a large number of data structures, many of which are irrelevant to the security sensitive operation, we need to collect the most relevant accesses and exclude noise. For example, a loop iterator is not interesting for our rule extraction, so we want to exclude it. Although all data structure accesses theoretically can be protected by a security check function, structure field accesses and global variable accesses are more commonly protected than short-lived local scalar variables. Therefore, we extract all structure field accesses and global variable accesses. In addition, a security sensitive operation, being an aggregate representation of its specific instances, is naturally represented by accesses to the *types* of data structures, and not by accesses to *specific* data objects. Thus, our rule inference engine considers structure types as opposed to actual objects.

4.2.4 How to infer rules?

Starting from the automatically identified root functions, we can extract the data structure access set for *each instance* of a security check function. To obtain the data structure access set protected by the security check function, we simply compute the *intersection* of all of these access sets. Since our static analysis can miss some data structure accesses for some root functions due to analysis imprecision, we do not require accesses to be protected by all instances. Instead, if intersecting an access set results in an empty set, we drop this access set because it is likely to be an incomplete set. As long as there are enough security check instances protecting the accesses, we are confident the accesses are security sensitive and the inferred rule is valid.

However, different from inferring general program rules, many security check functions are called only once or twice, which makes it difficult for the intersection strategy to be effective. We observed that many such functions are only called once or twice because Linux uses a centralized place to invoke such checks for different implementations. For example, check function `security_inode_rmdir()` is only called once in the virtual file system level, but it actually protects the sensitive rmdir operation of many different file systems. Therefore, semantically the check function is invoked once for *each* file system. Thus, we can intersect the rmdir operations of different file systems to obtain the essential protected sensitive accesses. This strategy makes it possible for AutoISES to automatically generate rules of reasonably small sizes with high confidence even for check functions that are called only a few times. This is realized by performing a function alias analysis and generating a separate static trace for each function alias, essentially treating each function alias as if it was a separate function call.

4.3 Our Static Analysis

We use *inter-procedural* and *flow-insensitive* static program analysis to infer security rules and detect violations. It is important to use inter-procedural analysis, because many sensitive data structure accesses related to the same sensitive operation are performed in different functions. In fact, these accesses can be many (e.g., 18) levels apart in the call chain, meaning the caller of one access can be the 18th ancestor caller of another access. An intra-procedural analysis would not adequately capture the security rules or be effective at detecting violations. In fact, without our inter-procedural analysis, we would not be able to detect almost any of the violations. For higher accuracy, we perform full inter-procedural analysis, which means that we allow our analysis tool to zoom into functions as deep as it can, i.e., until it has analyzed all reachable functions whose source code is available. We chose to use flow-insensitive analysis over flow-sensitive analysis because it is less expensive and scales better for large software.

As function pointers are widely used in Linux and Xen, we perform simple function pointer analysis by resolving a function pointer to functions with the same type. Our analysis is conservative in the absence of type cast.

5 Methodology and Results

We evaluated our tool on the latest versions, at the time of writing, of two large open source software, Linux and Xen. Table 2 lists their size information.

Software	Lines of Code	Total # of Check Functions
Linux	5.0M	96
Xen	0.3M	67

Table 2: Evaluated software. We excluded constructor and destructor type of security check functions from the list, because they are not authorization checks.

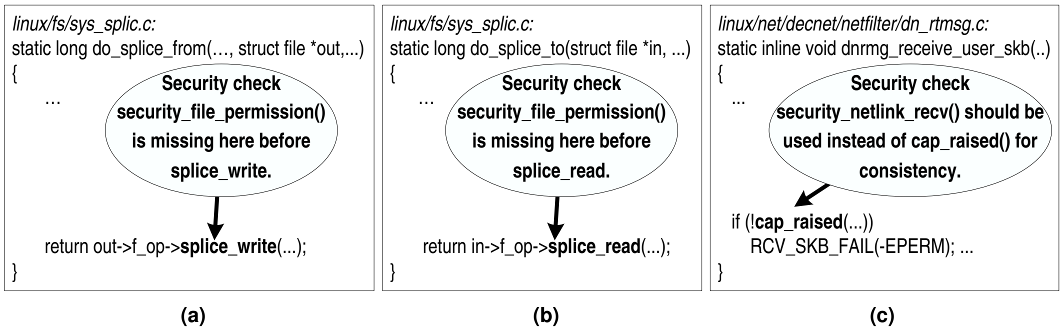
Table 3 shows our overall analysis and detection results. AutoISES automatically generated 84 code-level security rules, which served as the concrete security specifications of the two software we studied. These specifications are critical for verifying software correctness and security. Additionally they can help developers better understand the code and ease the task of software maintenance. We did not generate one rule for each security check mainly because some parts of the source code were not compiled for the default Linux kernel or Xen configuration, and were therefore not analyzed.

Based on our untrusted-space exposability study results, AutoISES reports violations that can be exposed to untrusted space as errors, and the others as warnings since they are less likely to be true security violations. Using the 84 automatically generated rules, AutoISES reported 8 error reports and 293 warning reports. A total of 8 true violations were found, 6 of which were from the error reports, and 2 were from the warnings reports (only the top warnings were examined). Among the 8 true violations, 7 of them have been confirmed by the corresponding developers. All of the automatically inferred rules were used by the AutoISES checker *directly* without being examined by us or the developers. If higher detection accuracy is desired, developers or tool users can examine rules before using them for violation detection.

These results demonstrate that AutoISES is effective at automatically inferring security rules and detecting violations to these rules, which closes an important gap in achieving security systems and greatly improves the practicality of security property checking and verification tools.

Software	# of rules	# of True Violations	False Positives in Errors	# of Warnings	
				# Inspected	Uninspected
Linux	51	7	1/6	25 (2)	265
Xen	33	1	1/2	3	0
Total	**84**	**8**	**2/8**	28 (2)	265

Table 3: Overall results of AutoISES. Numbers in parentheses are true violations in warning reports.



```
linux/fs/sys_splic.c:
static long do_splice_from(..., struct file *out,...)
{
    ...

    return out->f_op->splice_write(...);
}
```

```
linux/fs/sys_splic.c:
static long do_splice_to(struct file *in, ...)
{
    ...

    return in->f_op->splice_read(...);
}
```

```
linux/net/decnet/netfilter/dn_rtmsg.c:
static inline void dnrmg_receive_user_skb(..)
{
    ...

    if (!cap_raised(...))
        RCV_SKB_FAIL(-EPERM); ...
}
```

Security check security_file_permission() is missing here before splice_write.
Security check security_file_permission() is missing here before splice_read.
Security check security_netlink_recv() should be used instead of cap_raised() for consistency.
(a)
(b)
(c)

Figure 6: True violations AutoISES automatically detected in the latest versions of Linux kernel. All of these violations have already been confirmed by the Linux developers.

5.1 Detected Violations

We manually examined every error report and only the top warning reports (due to time constraints) to determine if a report is a true violation or a false positive.

5.1.1 True Violations

There are two types of true violations, exploitable violations and consistency violations.

Exploitable Violations Among the 8 true violations, 5 are exploitable violations. Figure 6 (a) and (b) show two exploitable violations. In Linux 2.6.21.5, security check `security_file_permission()` was missing before the file splice read and file splice write operation. Without the check, an unauthorized user could splice data from pipe to file and vice versa, which could cause permanent data loss, information leak, etc. This violation has already been confirmed by the Linux developers.

Consistency Violations We term the 3 remaining true violations *Consistency Violations*, meaning that although they may not be exploitable, they violate the consistency of using security check functions. Such inconsistencies can confuse developers and make the software difficult to maintain, both of which can contribute to more errors in the future. Therefore, it is important for developers to fix consistency violations.

Figure 6 (c) shows an example of a consistency violation. A security check `security_netlink_recv()`, which checks permission before processing the received netlink message, was missing in `dnrmg_receive_user_skb()`, which receives and processes netlink messages. This error could cause the kernel to receive messages from unauthorized users. However, `dnrmg_receive_user_skb()` did call function `cap_raised()`, which is what `security_netlink_recv()` calls eventually. In other words, it bypasses the security check interface functions, and calls the backend security policy functions, which is a bad practice and should be avoided.

At the time of writing, 2 out of the 3 consistency violations, including the example shown above, have been confirmed and fixed by the corresponding developers.

5.1.2 False Positives

The false positive rate in error reports is 2 out of 8. There are more false positives in the warning reports because no untrusted-space exposability analysis is performed on the warning reports. Developers can choose to focus on the error reports to save time, or also examine the warnings if desired.

Several factors can contribute to false positives. First, as we use conservative function pointer alias analysis, we can mistakenly consider accesses not related to an operation as part of the operation, and generate an imprecise rule. These extra accesses do not need to be protected by the security check, but our tool may still report such false violations. A static analysis tool with more advanced function pointer alias analysis could reduce such false positives.

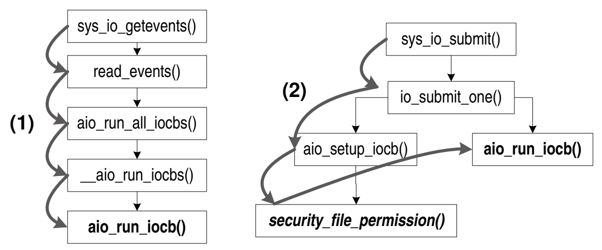


Figure 7: A false positive detected by AutoISES in Linux kernel 2.6.21.5. Only related functions are shown.

Additionally, certain semantics of the target code make some of the detected errors not exploitable. Figure 7 shows such an example where an implicit temporal constraint on certain system calls allows the coverage of a security check to span multiple system calls. AutoISES reported that a security check `security_file_permission()` should be called before `aio_run_iocb()`, but in the call chain in Figure 7(1) starting from a system call function `sys_io_getevents()`, the check `security_file_permission()` was missing. However, this is not an exploitable violation, because system call `sys_io_getevents()` cannot be called without system call `sys_io_submit()` being invoked first, which consults the proper security check in its callee function `aio_setup_iocb()` in call chain (2). Because AutoISES did not know this restriction in using the system calls, it reports the violation. However, if the file permission is changed after the setup system call `sys_io_submit()` and before the invocation of `sys_io_getevents()`, then unauthorized accesses can occur. Linux developers confirmed the potential of such violations, but are unlikely to fix it because the current Linux implementation does not enforce protection against this type of violations.

There are at least two ways to reduce or eliminate false positives. First, we can employ more accurate static analysis techniques. Additionally, as increasing granularity could reduce false positives (discussed later in Section 5.3), we can experiment with even finer granularity, such as distinguishing increment, decrement, and zeroing operations, to further reduce false positives.

5.2 Parameter Sensitivity and Time Overhead

By default, we set the threshold of $accessViolationCount$ to be 50% of the rule size, which is the total number of data structure accesses in a rule. We found that for Linux, the detection results are not very sensitive to this parameter, meaning that most true violations perform all or almost all of the data structure accesses, and false violations often perform none or only a few of the data structure accesses. These results show that the generated rules capture the implicit security rules well, and these rules are effective in helping detecting violations to them. For Xen, the results are more sensitive to the threshold. A possible explanation is that, in general Xen security checks are called fewer times compared to Linux kernel, therefore, there are fewer instances for AutoISES to learn precise rules. As a result, the inferred Xen rules contain more noisy accesses that do not need to be protected by the check functions. In this case, we set the threshold to be higher, 90%, to minimize the impact of noisy accesses.

AutoISES spent 86 minutes on inferring 51 rules from the entire Linux kernel, and 116 minutes on using these rules to check for violations in the entire Linux kernel. As the code size of Xen is much smaller, the time spent on Xen rule generation is 25 seconds, and 39 minutes for detection. This shows that our tool is efficient enough to be used in practice for large real world software.

5.3 Impact of Rule Granularity

In many cases, a coarse-grained rule is overly generalized and thus does not precisely represent the implicit security rules. For example, two different checks, `security_file_link()` and `security_file_unlink()` are designed to protect two different inode operations. However, as shown in Figure 8(a), the inferred operations of Granularity($F-$, $A+$) are the same. Using finer granularity, Granularity($F+$, $A+$), AutoISES is able to automatically infer two different operations (Figure 8(b)-(c)). For example, the unlink operation contains access `READ inode->i_size`, which is not part of the link operation.

Fine-grained rules cause less false positives during the detection stage. For 5 randomly selected security checks, compared with the most coarse-grained rules (Granularity($F-$,$A-$)) our most fine-grained rules (Granularity($F+$,$A+$)) on average cause 33% fewer false positives (in both error reports and warning reports). Granularity($F+$, $A-$) cause 20% fewer false positives, and Granularity($F-$, $A+$) 13.3% fewer. The results show that using finer granularity can greatly reduce the number of false positives, and adjusting the rule granularity could be considered as an important tuning parameter for other rule inference and violation detection tools [9, 11, 20, 22, 30].

Although coarse-grained rules produce a higher false positive rate, they can provide very useful information that fine-grained rules may fail to unveil. In the example above, the operation of Granularity($F-$, $A+$) is shared by almost all inode related security

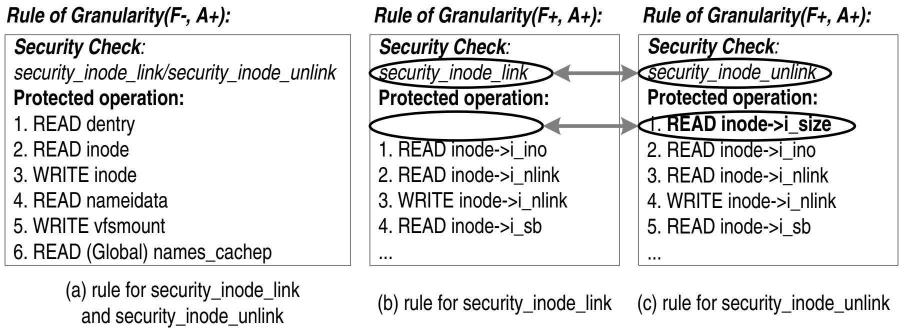


Figure 8: For two security checks, `security_file_link()` and `security_file_unlink()`, the inferred operations of Granularity($F-$, $A+$) are the same. If we use Granularity($F+$, $A+$), the learned operations are different, e.g., the unlink operation contains an extra access, `READ inode->i_size`.

check functions, including `security_inode_rename()`, `security_inode_rmdir()`, `security_inode_mkdir()`, `security_file_link()`, `security_file_unlink()`, `security_inode_symlink()`, etc. Therefore, the rule represent the common accesses of inode operations in general. A more fine-grained rule may fail to reveal the common behavior among all inode and file operations.

In addition, a fine-grained rule can be overly specific, and cause false negatives. We did not observe such cases for our most fine-grained rules in this study, i.e., our most fine-grained rules were able to detect all of the true violations. The result indicates that the default granularity we use is the best among the 4 levels of granularity in terms of detection accuracy, as they produce the least number of false positives, and the same number of false negatives as the coarse-grained rules. In the future, we plan to experiment with even finer granularity and its impact on both false positives and false negatives.

Results from different levels of granularity can be used as a metric for violation ranking. For example, a violation that is reported at all levels of granularity is probably more likely to be a true violation than one that is reported only at some levels. In our future work, we will explore using the number of granularity levels a violation occurs at to rank violations.

6 Discussions and Limitations

6.1 Key Techniques that Make AutoISES Work

Automatically generating security specifications poses several key challenges that make previous static analysis tools not directly applicable. We designed five important techniques (first four are new) to address these challenges as summarized below (Sec. column lists corresponding sections that describe the techniques):

Challenge	Our Solution	Sec.
How to represent a sensitive operation at the code level	Use *Data structure accesses* based on our key observation	1.2 & 3.2
High false positive rates as many sensitive operations can not be exposed to the untrusted space	Simple untrusted space exposability study to greatly reduce false positives	3.1
Root functions for analysis: Cannot simply start analysis from direct callers of a security check function	Automatic root function discovery: *Automatically* discover functions that actually use security check functions for authorization check	4.2.1
Insufficient invocation instances of security check functions	Leverage different implementations (e.g., from different file systems) of the same operation	4.2.4
Data structure accesses are spread in different functions.	Interprocedural analysis with function pointer analysis	4.3

6.2 Generalization

Although many of the solutions described above are designed for inferring security specifications and detecting security violations, some of the ideas are *general*, and can be applied to other applications. For example, our security rules are an important type of function-data correlation. Such function-data correlations widely exist in programs. Violating these implicit constraints results in buggy programs that may cause severe damage. Our techniques can be used to infer those general function-data correlations, e.g., a lock acquisition function required before accessing shared data structures, which can be used for detecting concurrency bugs. In addition, the strategy of using multiple implementations of the same virtual API to generate more precise rules is generally applicable to situations where source code at the virtual API level is not sufficient to generate reliable rules.

6.3 Limitations

False Negatives Similar to previous static analysis work, our approach can miss security violations. First, if a security check function is not invoked at all (e.g., `security_sk_classify_flow` is not used in Linux 2.6.11 yet) or the list of security check functions is incomplete, we would not be able to infer rules or detect violations related to these missing check functions.

Additionally, our analysis uses only data structure accesses to represent a security operation. Therefore, if the source code of such low-level accesses is not available, AutoISES will not be able to extract information about them, and the representation of the sensitive operation would be incomplete, potentially causing false negatives.

Moreover, AutoISES does not verify if the security check is performed on the same object as the sensitive operation. Therefore, if the proper security check is invoked, but on a different object, AutoISES will not detect this violation. Matching the actual object remains as our future work. Additionally, our flow-insensitive analysis could introduce false negatives. For example, if a security check is missing on a taken branch, but the check is invoked on the non-taken branch, AutoISES may not be able to detect the violation. Using a flow-sensitive analysis could address this problem.

Difficulty in Verifying Violations We manually examine error reports and warning reports to determine if a report is a true violation or a false positive. Unlike errors such as buffer overflows and null pointer dereferences, which are usually easy to confirm after the error is detected, the manual verification process for security violations is more difficult. To decide if a violation is exploitable, one needs to understand the semantics of the code, knowing what operations can interact with the untrusted space, such as the user space for Linux, and design a feasible way to exploit the attack. Conversely, to determine if a violation is a false positive, one needs to prove that either the operation is security insensitive, or that it is indeed covered by a security check that was not included due to analysis imprecision. Sometimes it requires deep knowledge of not only the target software, but also how the APIs are used by client software (e.g., the example discussed in Section 5.1.2). Such difficulties are mostly due to the inherent characteristics of security violations. However, we imagine that the task would be much easier for the original developers as they possess deep semantics knowledge of the code.

Non-authorization Checks A small number of security checks are not authorization checks, which do not protect any security operations. For example, `security_sk_free()` should be called *after* using a kernel sk buffer to clear sensitive data. Our current implementation does not support such rules where a security check function must be invoked after a certain operation. However, such rules can be easily supported by extending our current implementation to include the post-operation checks.

7 Related Work

Mining Security Sensitive Operations Ganapathy et al. used concept analysis to find fingerprints of security sensitive operations [15]. While both this approach and AutoISES try to map the high level security sensitive operation (e.g., rmdir) to its implementation (e.g., the C code sequences that actually perform the remove directory operation), there are two major differences. First, the goals and assumptions are different. We aim to identify the pairing relationship between a security check and the code level representation of the sensitive operation that the check guards. Thus we assume the code already implements a reference monitor and is mostly correct; our goal is therefore to discover cases where the reference monitor is bypassed. Ganapathy's goal, on the other hand, is to retrofit code with security. Thus they assume that the code does not have security built in. Rather, they need to identify sequences of code that represent a unit of security sensitive operation and that should be guarded by a security hook. In order to do that they need more prior knowledge with regard to the API and the security sensitive data structures. In our case, all information except the list of security check functions, and the list of system call functions and hypercall functions, is inferred from the code itself. Second, while our inferred operations are used directly by our checker without being examined manually, their operations still require manual refinement prior to use.

Although automatic hook placement is promising, it has not been adopted in reality yet. Therefore, while we should encourage automatic hook placement, it is still highly desirable to seek alternative, complementary solutions that can automatically infer security rules from existing or legacy source code and detect security vulnerabilities.

Detection and Verification Tools The past years have seen a proliferation of program analysis and verification tools that can be used to detect security vulnerabilities or verify security properties [2, 4, 5, 6, 9, 12, 14, 16, 18, 27, 30]. However, no previous work can automatically generate code-level security specifications and instead require developers or users to provide these specifications. Previous work [30] takes *manually* identified simple security rules to check for security vulnerabilities. As discussed in details in Section 1, the rules are coarse and imprecise, resulting in many false alarms. Additionally, the approach can potentially fail to detect cases where the check and the operation does not match because the rules do not specify which check is required for which operation. Edwards et al. dynamically detect inconsistencies between data structure accesses to identify security vulnerabilities [9]. While a dynamic approach is generally more accurate, it suffers from coverage problem - only code that is executed can be analyzed. In addition, it requires manually written filtering rules to guide the trace analysis in order to detect security violations.

Inferring Programming Rules Several techniques have been proposed to infer different types of programming rules from source code or execution trace [3, 10, 11, 20, 22, 24]. As already discussed in Section 1, previous techniques is not directly applicable to our problem, because they are limited by the types of rules they can infer. Specifically, Engler et al. extract programming rules based on several manually identified rule templates, such as function $\langle A \rangle$ and $\langle B \rangle$ should be paired, function $\langle F \rangle$ must be checked for failure, and null pointer $\langle P \rangle$ should not be dereferenced [10]. PR-Miner focuses on inferring correlations among *functions* [20]. Variable value related program invariants are inferred by Daikon [11], and MUVI[22] infers *variable-variable* correlations for detecting multi-variable inconsistent update bugs and multi-variable concurrency bugs. A few other approaches infer API and/or abstract data type related rules[3, 24]. Different from all these studies, we infer rules related to security *functions protecting a group of data structure accesses* based on our key observation. Inferring different types of rules requires different techniques. In addition, dynamic analysis is used in [3, 11], therefore the coverage is limited because only instrumented and executed code is used for rule learning. Moreover, unlike PR-Miner which uses only intraprocedural analysis, our analysis is interprocedural, which is one of the key techniques that allow us to infer complicated and detailed security rules. Additionally, while PR-Miner uses more complex data mining techniques to infer programming rules, we leverage readily available prior knowledge about part of our rules, the security check functions, so that we can extract security rules without expensive data mining techniques.

Inferring Models and Rules in General The general idea of automatically extracting models from low-level implementation has been discussed in previous literature [8, 17, 21]. For example, Lie et al. proposed automatic extraction of specifications from actual protocol code and then running the extracted specifications through a model checker [21]. While conceptually these approaches bear some resemblance to the approach taken by AutoISES, we are the first to show the feasibility of automatic extraction of security specifications from actual implementation. In addition, none of the previous tools have demonstrated the ability to scale to programs the size of the Linux kernel.

Lee et al. [19] use data mining techniques to learn intrusion detection model for adaptive intrusion detection. Tongaonkar et al. [26] infer high-level security policy from low level firewall filtering rules. None of these work infer access control related security rules.

8 Conclusions and Future Work

This paper makes two contributions. One is to automatically infer code-level security rules and detect security violations. Our tool, AutoISES, automatically inferred 84 security rules from the latest versions of Linux kernel and Xen, and used them to detect 8 security vulnerabilities, demonstrating the effectiveness of our approach. The second contribution is to take the first step to quantitatively study the impact of the rule granularity on rule generation and verification. This approach is orthogonal to our first contribution, and can be applied to other rule inference tools.

While this work focuses on rule inference and violation detection in Linux kernel and Xen, our techniques can be used to generate rules and detect violations in other access control systems. In addition, the techniques can be applied to infer general function-data correlation type of rules, such as lock acquisition functions protecting shared variables accesses. In the future, we plan to improve our analysis and detection accuracy by employing a more advanced static analysis tool and using finer rule granularity.

9 Acknowledgments

We greatly appreciate the anonymous reviewers for their feedback. We thank Mihai Christodorescu, Brett Daniel, Darko Marinov and Vugranam Sreedhar for their invaluable comments and observations during this work. This research is supported by NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), DOE Early Career Award DE-FG02-05ER25688, and Intel gift grants.

References

[1] Vulnerability summary CVE-2006-1856. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-1856.

[2] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering*, 2007.

[3] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, 2002.

[4] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, 2002.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *Proceedings of the 11th International Static Analysis Symposium*, 2004.

[6] 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*, 2002.

[7] G. S. Coker. Xen security modules: Intro. http://lists.xensource.com/archives/html/xense-devel/2006-09/msg00000.html.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[9] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[10] D. R. 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*, 2001.

[11] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[12] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[13] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[14] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[15] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[16] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[17] G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. *Software Testing, Verification and Reliability*, 2001.

[18] T. Jaeger, A. Edwards, and X. Zhang. Consistency analysis of authorization hook placement in the linux security modules framework. *ACM Transactions on Information and System Security*, 2004.

[19] W. Lee, S. J. Stolfo, and K. W. Mok. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 2000.

[20] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[21] D. Lie, A. Chou, D. Engler, and D. L. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[22] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[23] NSA. Security-Enhanced Linux (SELinux). Available at http://www.nsa.gov/selinux.

[24] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 174–184.

[25] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 1995.

[26] A. Tongaonkar, N. Inamdar, and R. Sekar. Inferring higher level policies from firewall rules. In *Proceedings of the 21st Large Installation System Administration Conference*, 2007.

[27] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 2000 Network and Distributed Systems Security Conference*, 2000.

[28] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[29] C. Wysopal and C. Eng. Static detection of application backdoors. Available at http://www.veracode.com/images/stories/static-detection-of-backdoors-1.0.pdf.

[30] X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

Notes

[1] Portions of the work performed while Lin Tan was a summer intern at IBM Research.