Automatic Discovery of API-Level Vulnerabilities

Vinod Ganapathy[†], Sanjit A. Seshia[‡], Somesh Jha[†], Thomas W. Reps[†], Randal E. Bryant[‡]
[†]Computer Sciences Department, University of Wisconsin-Madison
[‡]School of Computer Science, Carnegie Mellon University
{vg|jha|reps}@cs.wisc.edu {sanjit|bryant}@cs.cmu.edu

UW-MADISON COMPUTER SCIENCES TECHNICAL REPORT: UW-CS-TR-1512, JULY 2004.

Abstract

A system is vulnerable to an API-level attack if its security can be compromised by invoking an allowed sequence of operations from its API. We present a formal framework to model and analyze APIs, and develop an automatic technique based upon bounded model checking to discover API-level vulnerabilities. If a vulnerability exists, our technique produces a trace of API operations demonstrating an attack. Two case studies show the efficacy of our technique. In the first study we present a novel way to analyze `printf`-family format-string attacks as API-level attacks, and implement a tool to discover them automatically. In the second study, we model a subset of the IBM Common Cryptographic Architecture API, a popular cryptographic key-management API, and automatically detect a previously known vulnerability.

1 Introduction

Software modules communicate through application programming interfaces (APIs). Failure to respect an API's usage conventions or failure to understand the semantics of an API may lead to security vulnerabilities. For instance, Chen *et al.* [13] demonstrated a security vulnerability in `sendmail-8.10.1` that was due to a misunderstanding of the semantics of UNIX user-id setting commands. A programmer mistakenly assumed that setuid(getuid()) would always drop all privileges. Prior work has addressed automatic detection of erroneous API-usage patterns, such as ordering constraints when invoking system calls or when acquiring locks, in client software modules [2, 3, 4, 12, 23]. These projects demonstrate that care needs to be exercised when using APIs. In particular, the *semantics* of the API should be developed carefully and should be exposed to users of the API. In addition, *rules of usage*, such as ordering restrictions, must be respected when the API is used in client software modules.

A natural question that arises is whether API-level vulnerabilities can be discovered automatically. We have developed a technique that analyzes specifications of APIs to identify sequences of API operations that take a system from a good state to a state that violates a given safety property. In some cases we can prove that the API is secure with respect to the property in question. We make the following contributions:

- We develop a formal framework that can capture the semantics of APIs and sequences of API operations allowed by a system.
- We develop a technique based upon bounded model checking that analyzes API specifications and automatically identifies API-level vulnerabilities. If a vulnerability exists, the technique produces an attack, i.e., a sequence of API operations exploiting the vulnerability. Descriptions of the framework and the technique appear in Section 3.
- We demonstrate the efficacy of our technique through two case studies from different problem domains. In Section 4, we show a novel way to analyze and understand `printf`-family format-string attacks [27, 38]. We model these as API-level attacks and show how our technique discovers them automatically. In Section 5, we apply the technique to the IBM Common Cryptographic Architecture (CCA) API [29] and discover a previously known vulnerability [7]. In each of the above cases we have developed prototype tools.

A direct consequence of the technique is the capability to infer rules for safe API-usage. Such rules can benefit both static and dynamic analysis tools. Tools such as MOPS [12], meta-level compilers [3, 23], and SLAM [4] statically analyze source code for deviations from acceptable patterns of API-usage. However, the patterns checked are typically produced manually. Our analysis could benefit such tools by automating the generation of such patterns.

The rules could also be used to create reference monitors [41] that track the runtime behavior of applications and enforce API-usage rules. We discuss these applications in Section 6.

2 Overview

Our automated technique to discover API-level vulnerabilities is based upon *bounded model checking* [6, 15]. A schematic overview of the technique is shown in Figure 1. The bounded model checker accepts the description of a system and its API, specified using our framework, which has four components: (1) a set of variables that describe the state of the system, (2) the initial state of the system, (3) the set of API operations allowed and the semantics of these operations in terms of how they change the state of the system, and (4) a representation of the set of traces of API operations allowed by the system.

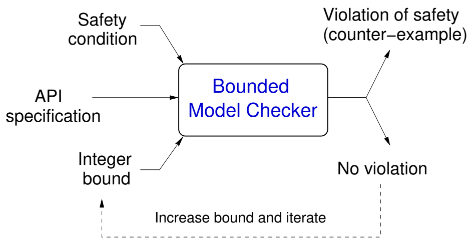

Figure 1: A schematic overview of the method.

For instance, consider a toy protection system as described by Harrison *et al.* [26]. A protection system is defined by a finite set of rights and commands, and its state is given by a triple (*S, O, P*), where *S* is a set of subjects, *O* is a set of objects, and *P* is an access matrix with a row for each subject and a column for each object. As presented by Harrison *et al.*, each subject is also an object, and we have $S \subseteq O$. The entry $P[s, o]$ of the access matrix is a set of rights that subject s has on object o. We restrict ourselves to three rights, *own*, *read*, and *write*, with their natural meanings.

The initial state of the protection system is given by the initial values of *S*, *O*, and *P*. Assume that these values are $S = O = \{\mathsf{A}, \mathsf{B}\}$, $P[\mathsf{A}, \mathsf{A}] = P[\mathsf{B}, \mathsf{B}] = \{own, read, write\}$, and $P[\mathsf{A}, \mathsf{B}] = P[\mathsf{B}, \mathsf{A}] = \emptyset$. In other words, A and B have all possible rights upon themselves, but no rights on each other.

The commands presented by the protection system define the API; each command changes the state of the protection system. We restrict ourselves to the three commands shown below with their semantics. We also assume that the protection system allows these operations to be applied in any order.

1. Create(s, o): If $s \in S$ and $o \notin O$, adds o to O, creates a new column o in P and enters *own* into $P[s, o]$.
2. Confer$_{read}(s_1, s_2, o)$: If $s_1, s_2 \in S$ and $o \in O$, enters *read* into $P[s_2, o]$ if $own \in P[s_1, o]$.
3. Confer$_{write}(s_1, s_2, o)$: If $s_1, s_2 \in S$ and $o \in O$, enters *write* into $P[s_2, o]$ if $own \in P[s_1, o]$.

In addition to a description of the protection system, the bounded model checker also accepts a safety condition. For instance, the safety condition could be a security policy such as "no subject can both read and write to an object that it does not own". The model checker systematically explores all allowed traces of API operations up to a certain length, given by an integer bound, and determines whether each trace satisfies the safety condition. If the model checker finds a violation of the safety policy, it terminates with a trace of API operations showing the vulnerability. For instance, a bound of at least 3 discovers the following API-level vulnerability in the protection system: Create(A, `file`) $\rightarrow$ Confer$_{read}$(A, B, `file`) $\rightarrow$ Confer$_{write}$(A, B, `file`). This sequence of API operations adds (B, `file`, *read*), and (B, `file`, *write*) to *P*, but does not add (B, `file`, *own*), and this violates the safety condition, because B does not *own* `file`, but can *read* and *write* it.

If the model checker terminates without a counter-example, we must increase the bound and iterate. In Section 3, we note that it is undecidable to check if an arbitrary system is vulnerable to API-level attacks. Thus, in general, the iterative process could go on forever. Our procedure is *sound*, but *incomplete*—any vulnerabilities found will indeed be exploitable in the model; however, it is not always possible to discover all vulnerabilities. In certain cases, including the study in Section 4, it is possible to derive values of the bound for which the procedure is complete.

3 Formal Framework

We present a formal framework to model and analyze APIs. An API is the interface that a system presents to client modules. Each command in the API changes the state of the system in a predefined way and hence is a *state transformer*. A sequence of API operations defines a state transformer obtained by composing the state transformers of the individual API operations. We focus on such sequences of API operations, and how they affect the security of the underlying system.

Formally, a *system* $\mathcal{S}$ is defined by a quadruple ($\mathcal{V}$, **Init**, Σ, $\mathcal{L}$):

- $\mathcal{V}$ denotes a finite set of variables $\{v_1, v_2, \ldots, v_n\}$ where $v_i \in \mathcal{D}_i$ for some (possibly infinite) domain of values $\mathcal{D}_i$. The value of the vector $\vec{x} = (v_1, v_2, \ldots, v_n)$ provides an instantaneous description (henceforth referred to as *state*) of the system $\mathcal{S}$. Note that $\vec{x} \in \mathcal{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$.
- **Init**: $\mathcal{D} \rightarrow$ BOOL is a predicate that characterizes the initial states of the system. Each vector $\vec{x}$ such that **Init**($\vec{x}$) holds is a possible initial state of the system.
- Σ denotes a finite set of API operations $\{\mathsf{op}_1, \mathsf{op}_2, \ldots, \mathsf{op}_m\}$. Each operation op_i may also take some input parameters, denoted by the vector $\vec{a}_i$, from some domain $\mathcal{A}_i$. Each op_i defines a family of state-transformation functions: $\mathsf{op}_i(\vec{a}_i)$: $\mathcal{D} \rightarrow \mathcal{D}$. The semantics of $\mathsf{op}_i(\vec{a}_i)$ is given by predicates that define its pre- and post-conditions, $\mathbf{Pre}_i(\vec{a}_i)$: $\mathcal{D} \rightarrow$ BOOL and $\mathbf{Post}_i(\vec{a}_i)$: $\mathcal{D} \rightarrow$ BOOL, as follows: Suppose the state of $\mathcal{S}$, denoted by $\vec{x}$, is such that $\mathbf{Pre}_i(\vec{a}_i)(\vec{x})$ holds, then $\mathsf{op}_i(\vec{a}_i)(\vec{x}) = \vec{y}$ and $\mathbf{Post}_i(\vec{a}_i)(\vec{y})$ holds. If $\mathbf{Pre}_i(\vec{a}_i)(\vec{x})$ does not hold, then $\mathsf{op}_i(\vec{a}_i)$ leaves the state of the system unchanged.
- $\mathcal{L} \subseteq \Sigma^*$ is a language of API operations.

In addition, a predicate **Bad**: $\mathcal{D} \rightarrow$ BOOL defines the set of bad states; each vector $\vec{x}$ such that **Bad**($\vec{x}$) holds is a state that the system should never enter. **Bad** is defined based upon the security properties required for $\mathcal{S}$.

$\mathcal{L}$ plays two roles in the framework:

1. It encodes temporal restrictions on API operations that are inherent in system $\mathcal{S}$, as could be specified using a reference monitor. This allows us to avoid checking sequences of API operations that are explicitly disallowed by $\mathcal{S}$, and thus reduce false alarms.
2. It formalizes the possible set of traces of API operations that could be generated by a client. This avoids wasteful exploration of the state space during verification, and also reduces false alarms. For instance, suppose the API in question is the set of system calls supported by an operating system, and that we wish to verify that an application that uses system calls conforms to a safety property and does not launch an API-level attack on the operating system. Rather than considering all interleavings of arbitrary system calls, it is sufficient to restrict our attention to system call traces that can be generated by the application.

It is also possible to handle systems in which both kinds of restrictions are important. To do so, $\mathcal{L}$ is defined as the intersection of two trace languages, one that plays the first role, and one that plays the second.

A language recognizer $\mathcal{R}$ for $\mathcal{L}$ is a machine that accepts a string of API operations and determines whether it is a member of $\mathcal{L}$ or not. In general, a recognizer need not exist for $\mathcal{L}$. We restrict ourselves to cases where a recognizer $\mathcal{R}$ exists, for instance, when $\mathcal{L}$ is regular or context-free. For the case studies in Sections 4 and 5, we consider a special case of the framework presented above. In particular, $\mathcal{L}$ will be a regular language and its recognizer will be a finite-state machine called the *API-automaton*.

Definition 1 (API-safety) Consider a system $\mathcal{S}$ whose state $\vec{x}$ satisfies **Init**($\vec{x}$). For a predicate **Bad**, $\mathcal{S}$ is said to be *safe* with respect to $\vec{x}$ if $\mathcal{L}$ does not contain a trace of API operations that takes $\mathcal{S}$ into a state $\vec{y}$ that satisifes **Bad**. Formally, $\mathcal{S}$ is safe with respect to $\vec{x}$ if there is no satisfying assignment to the formula:
$\exists \mathsf{op}_{i_1}, \mathsf{op}_{i_2}, \ldots, \mathsf{op}_{i_k}, \vec{a}_1, \vec{a}_2, \ldots, \vec{a}_k.\ (\mathsf{op}_{i_1} \cdot \mathsf{op}_{i_2} \cdot \ldots \cdot \mathsf{op}_{i_k} \in \mathcal{L}) \wedge \mathbf{Bad}(\mathsf{op}_{i_k}(\vec{a}_k) \circ \ldots \circ \mathsf{op}_{i_2}(\vec{a}_2) \circ \mathsf{op}_{i_1}(\vec{a}_1)\ (\vec{x}))$
for any finite value of k. (Here '$\cdot$' denotes concatenation, and '$\circ$' denotes function composition.)

Note that only sequences of API operations in $\mathcal{L}$ are checked for safety. We have the following fact, whose proof follows from a similar theorem for protection systems [26].

Fact 1 (Undecidability) Given an arbitrary system $\mathcal{S}$ and a predicate **Bad**, no algorithm can decide if $\mathcal{S}$ is safe with respect to state $\vec{x}$.

The above fact is obvious if a recognizer $\mathcal{R}$ for $\mathcal{L}$ does not exist. However, the API-safety problem remains undecidable even if we impose the restriction that $\mathcal{L}$ is a regular language. This is because one or more of the domains $\mathcal{D}_i$ could be infinite. If the domains $\mathcal{D}_i$ are all finite, and $\mathcal{L}$ is regular or context-free, then it is possible to check that a system is safe with respect to an initial state $\vec{x}$. In our case studies, we do not make any assumptions on the finiteness of the domains $\mathcal{D}_i$.

3.1 Solution Methodology

As a consequence of Fact 1, we can only propose semi-algorithms for the API-safety problem. Our approach is based on bounded model checking.

Bounded model checking was originally developed for finding logical errors in finite-state transition systems [6, 15]. The inputs to such a model checker consist of a finite-state transition system M, a logical formula ψ to be checked, and a natural number N. The model-checking procedure checks if there are sequences in M of length less than or equal to N that violate ψ. This process proceeds by encoding computation sequences and ψ as a decision problem, such as satisfiability of a Boolean formula, and invoking a decision procedure, such as a Boolean satisfiability (SAT) solver. Recent advances in SAT solving (e.g., [37]) have made bounded model checking a practical tool for analyzing finite-state systems. This procedure is sound, so counter-examples found will indeed be vulnerabilities in the model. It is not complete, because sequences of length greater than N are not checked. However, for finite-state systems it is often possible to derive a value of N for which the procedure is complete [17].

Many real-world systems are infinite-state. In particular, the set of variables $\mathcal{V}$ that describe the state of the system could be from infinite domains, such as integers. Bounded model checking has also been used to analyze infinite-state systems for correctness. The approach works exactly as in the finite-state case, except that a decision procedure for a more expressive logic is used in place of a SAT solver. The general problem of model checking infinite-state systems is undecidable; however, bounded model checking for a fixed bound N is decidable when the satisfiability problem in the underlying logic is decidable. In recent times, bounded model checking for infinite-state systems has been made possible by the development of efficient decision procedures for expressive, decidable logics (e.g., [10, 20, 48]) that leverage advances in SAT solving. Section 4 considers a case where we use bounded model checking to identify API-level vulnerabilities in an infinite-state system.

Our approach to the API-safety problem restricts attention to sequences of API operations in $\mathcal{L}$ of length less than or equal to a bound N. Thus, for a state $\vec{x}$, we systematically check all sequences of operations $\mathsf{op}_{i_1} \ldots \mathsf{op}_{i_k} \in \mathcal{L}$ such that $k \leq N$ to determine if $\mathtt{Bad}(\mathsf{op}_{i_k}(\vec{a}_k) \circ \ldots \circ \mathsf{op}_{i_1}(\vec{a}_1)(\vec{x}))$ is satisfied for any values of $\vec{a}_k, \ldots, \vec{a}_1$. This is accomplished using a bounded model checker that accepts a description of the system $\mathcal{S}$ and the bound N. If $\mathcal{L}$ is regular, then it is described using an API-automaton. The model checker explores all execution paths of length less than or equal to N in the API-automaton, checking, for each state on that path, if **Bad** is satisfied. For cases where $\mathcal{L}$ is context-free, pushdown model checkers [42] will have to be used. In this paper, we only consider cases where $\mathcal{L}$ is regular, and $\neg$**Bad** is a safety property.

3.2 Illustrative Example

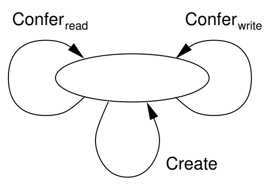

Figure 2: API-automaton for $\mathcal{L}$.

We illustrate the concepts developed above using the protection system example from Section 2. Recall that in the example, we initially had two subjects and objects, A and B. In our framework, we have $\mathcal{S}$ = ($\mathcal{V}$, **Init**, Σ, $\mathcal{L}$), where

- $\mathcal{V} = \{S, O, P\}$. Note that all three variables are set-valued, because the matrix P can also be viewed as a set of triples (s, o, r), where r denotes a right.
- **Init** = ($S = O = \{\mathsf{A}, \mathsf{B}\}$) $\wedge$ ($P[\mathsf{A}, \mathsf{A}] = P[\mathsf{B}, \mathsf{B}] = \{own, read, write\}$) $\wedge$ ($P[\mathsf{A}, \mathsf{B}] = P[\mathsf{B}, \mathsf{A}] = \emptyset$).
- $\Sigma = \{\mathsf{Create}, \mathsf{Confer}_{read}, \mathsf{Confer}_{write}\}$. The predicate **Pre**($s$, o) for Create(s, o) asserts that such an entry does not already exist in P, while **Post**(s, o) asserts that an entry (s, o) is created in P and $own \in P[s, o]$. The predicate **Pre**(s_1, s_2, o) for $\mathsf{Confer}_{read}(s_1, s_2, o)$ asserts that $own \in P[s_1, o]$, and **Post**(s_1, s_2, o) asserts that $read \in P[s_2, o]$. The predicates for Confer_{write} are similar.
- $\mathcal{L} = \Sigma^*$. That is, all possible interleavings of the API operations are permitted in this example.

To verify that "no subject can both read and write to an object that it does not own", we use the predicate **Bad** = $\exists s,o.\ (s \in S) \wedge (o \in O) \wedge (read, write \in P[s, o]) \wedge (own \notin P[s, o])$. The API-automaton for $\mathcal{L}$ for the protection system is shown in Figure 2. Bounded model checking for this case is equivalent to "unrolling" this API-automaton a finite number of times and checking that the property holds. When presented with a bound of at least 3, a bounded model checker explores the path $\mathsf{Create} \rightarrow \mathsf{Confer}_{read} \rightarrow \mathsf{Confer}_{write}$, and discovers the vulnerability.

3.3 Case studies

We demonstrate applications of the technique to two real world problems. In Section 4, we formalize `printf`-family format-string vulnerabilities, and build a tool to discover format-string attacks automatically. In Section 5, we model a subset of the IBM Common Cryptographic Architecture API and discover a previously known vulnerability.

4 `Printf`-family Format-String Vulnerabilities

Format-string vulnerabilities [27, 38] are a dangerous class of bugs that allow an attacker to execute arbitrary code on the victim machine. `printf` is a variable-argument C function that treats its first argument as a *format-string*.[1] The format-string contains *conversion specifications*, which are instructions that specify the types that `printf` should assign to other arguments, and instructions on how to format the output. For instance, the conversion specification `"%s"` instructs `printf` to look for a pointer to a `char` value as its next argument, and print the value at that location as a string. When `arg` does not contain conversion specifiers, the statements `printf("%s", arg)` and `printf(arg)` have the same effect. However, if `printf(arg)` is used in an application, and a user can control the value passed to `arg`, then the application is susceptible to a format-string vulnerability. Shankar *et al.* [45] have proposed a technique to analyze source code to identify "tainted" format-strings that can be controlled by an attacker. Potentially vulnerable `printf` locations can also be identified in binary executables [27]. A possible fix for such vulnerabilities is to do a source-to-source transformation that replaces all occurrences of `printf(arg)` with `printf("%s", arg)`, but this may not always be possible, for instance when the source code of the application is not available, or when the application generates format-strings dynamically. The tools mentioned above are also incapable of producing a format-string that demonstrates the attack at the vulnerable locations they identify.

We present a novel way to analyze and understand `printf`-family format-string vulnerabilities. We treat the format-string as a sequence of commands that instructs `printf` to look for different types of arguments on the application's runtime stack. We have built a tool that can analyze potentially vulnerable call sites to `printf` and determine if an attack is possible. If an attack is possible, our tool produces a format-string that demonstrates the attack. The technique does not require the source code of the application and can analyze potentially vulnerable `printf` locations from binary executables. Our technique could potentially be used in conjunction with the tool proposed by Shankar *et al.* to identify format-strings that exploit the vulnerabilities identified, thus confirming the presence of attacks. Our discussion and implementation make the following platform-specific assumptions, although the technique applies to other platforms as well:

1. We work with the x86 architecture. In particular, the runtime stack of an application grows from higher addresses to lower addresses, and the machine is assumed to be little-endian.
2. The arguments to a function are pushed on the stack from right to left. A call to foo(arg_1, arg_2) pushes arg_2 on the stack followed by arg_1. This is a popular C calling convention that is implemented by several compilers.
3. We analyze `printf` as implemented in `glibc-2.3`.

4.1 Understanding `printf`

```
(1) int foo (char *usrinp) {
(2)   char fmt[LEN];
(3)   int a, b;
(4)   ...
(5)   strncpy(fmt, usrinp, LEN - 1);
(6)   printf(fmt);
(7)   ...
(8) }
```

Figure 3: A procedure with a vulnerable call to `printf`

This section reviews how `printf` works. Consider the code fragment shown in Figure 3. Procedure `foo` accepts user input, which is copied into the local variable `fmt`, a local array of `LEN` characters. `printf` is then called with `fmt` as its argument. Because the first argument to `printf` can be controlled by the user, this program can potentially be exploited. When `printf` is called on line (6), the arguments passed to `printf` are placed on the stack, the return address and frame pointer are saved, and space is allocated for the local variables of `printf`, as shown in Figure 4(A). In this case, `printf` is called with a pointer to `fmt`, which is a local character buffer in `foo` shown as the heavily shaded region in Figure 4(A). Hence the value of the

[1] We restrict our attention to `printf`; the concepts extend to other `printf`-family functions in a natural way.

first argument contains the starting address of fmt.

As mentioned earlier, printf assigns special meaning to the first argument passed to it, and treats it as a format-string. Any other arguments passed to printf appear at higher addresses than the format-string on the runtime stack. In our case, only fmt was passed as an argument, and hence there are no other arguments on the runtime stack.

The printf implementation internally maintains two pointers to the stack; we refer to these pointers as FMTPTR and ARGPTR. The purpose of FMTPTR is to track the current formatting character being scanned from the format-string, while ARGPTR keeps track of the location on the stack from where to read the next argument. Before printf begins to read any arguments, FMTPTR is positioned at the beginning of the format-string and ARGPTR is positioned just after the pointer to the format-string fmt, as shown in Figure 4(A).

When printf begins to execute, it moves FMTPTR along format-string fmt. Advancing a pointer makes it move towards higher addresses in memory, hence FMTPTR moves in the direction opposite to which the stack grows. The printf system can be in one of two "modes". In *printing* mode, it reads bytes off the format-string and prints them. In *argument-capture* mode, it reads arguments from the stack from the location pointed to by ARGPTR. The type of the argument, and thus the number of bytes by which ARGPTR has to be advanced as it reads the argument, is determined by the contents of the location pointed to by FMTPTR. As FMTPTR and ARGPTR move toward higher addresses, they reach intermediate configurations, as shown in Figure 4(B). Note that ARGPTR advances only if the contents of fmt causes printf to enter argument-capture mode at least once.

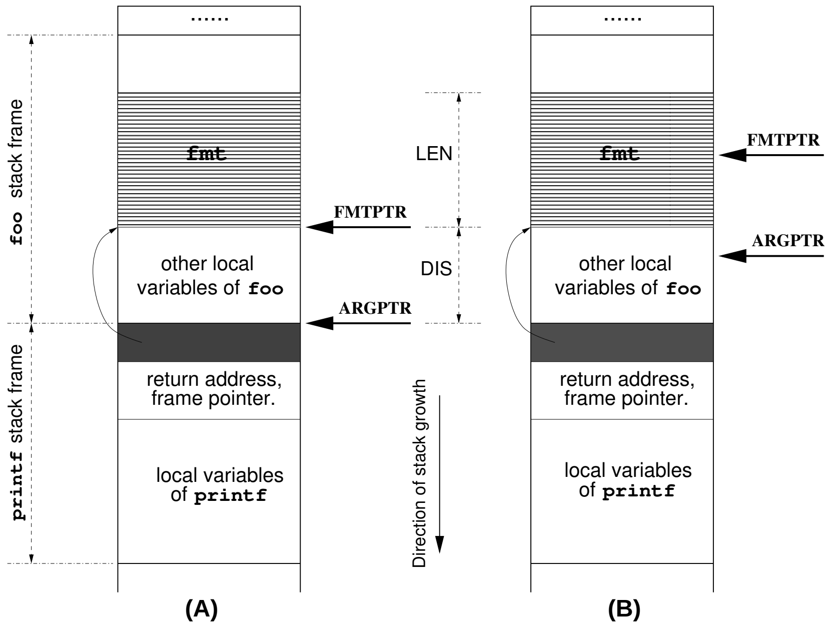

Figure 4: The runtime execution stack for the program in Figure 3.

To take a concrete example, suppose that fmt is "Hi%d" when printf is called in Figure 3. printf starts off in printing mode, and advances FMTPTR, printing Hi to stdout as a result. When FMTPTR encounters the byte "%", it enters argument-capture mode. When FMTPTR is advanced, it points to the byte "d" – which instructs printf to read four bytes from the location pointed to by ARGPTR and print the resulting value to the terminal as an integer. This also results in ARGPTR being advanced by four bytes, the size of an integer. Note that no integer arguments were explicitly passed to printf in Figure 3, hence instead of reading a legitimate integer value off the stack, in this case ARGPTR reads the values of local variables in the stack frame of foo. As a result, it is possible to read the contents of the stack, which may possibly contain values of interest to an attacker, such as return addresses.

In the format-string attacks that we consider, the goal of the attacker is to control the contents of the format-string in such a way that ARGPTR advances along the stack until it enters the format-string itself. By doing so, the attacker can control the arguments read by printf. We develop this point further in Section 4.3.

4.2 Formalizing **printf**

Observe that each byte in the format-string can be interpreted as an instruction to `printf` to move FMTPTR and ARGPTR by an appropriate amount. These bytes also instruct `printf` as to the types of the arguments passed to it. Hence, in our formulation, each byte in the format-string is treated as an API-command to `printf` and thus the format-string is a sequence of API operations. Our goal is to discover possibly malicious sequences — which corresponds to finding format-strings that can be used for an attack.

Each `printf` call is characterized by two parameters, namely the values DIS and LEN shown in Figure 4. Format-string vulnerabilities occur only when the format-string that can be controlled by the attacker is a buffer on the runtime stack. LEN denotes the length of this buffer. DIS denotes the number of bytes that separate the pointer to the format-string from the format-string itself. Figure 4 only shows a simple scenario where the stack frame containing the format-string and the stack frame containing the pointer to it are adjacent. In general, they can be separated by stack frames of several intermediate functions, resulting in larger values of DIS. Note that the values of DIS and LEN are sufficient to capture the important details of the problem. Moreover, the values of DIS and LEN for each `printf` call can be obtained by disassembling the binary executable file of the application that calls `printf`, and examining the call graph and the size of stack frames of functions.

Formally, the `printf` system is defined as $\mathcal{S} = (\mathcal{V}$, **Init**, Σ, $\mathcal{L})$, where:

- $\mathcal{V}$ denotes the set of local variables used by the implementation of `printf` that capture the state of the `printf` system. We identified 24 local variables (or "flags") with integer and Boolean values[2] by examining the source code of `printf`. While our implementation considers all of these flags, for purposes of explanation we restrict ourselves to just four flags, namely, FMTPTR, ARGPTR, DONE, and IS_LONGLONG. FMTPTR and ARGPTR are pointers whose functionality was discussed earlier. We shall treat these as integer values. DONE is an integer that counts the number of bytes printed, and IS_LONGLONG is a Boolean variable that determines whether the argument on the stack is a `long long` value or not (a `long long int` is 8 bytes in length).
- **Init**: The initial state of `printf` is determined by the values to which the flags in $\mathcal{V}$ are initialized. We assume that all addressing is relative to the initial location of ARGPTR, and hence **Init** is defined as $(\text{ARGPTR} = 0) \wedge (\text{FMTPTR} = \text{DIS}) \wedge (\text{DONE} = 0) \wedge (\text{IS\_LONGLONG} = \text{FALSE})$.
- Σ: As explained, each byte in the format-string is interpreted as an instruction to the `printf` system. Hence Σ is $[0..255]$, i.e., all possible byte values. The values of **Pre** and **Post** for each operation are based on how it changes the state of `printf`, and were obtained by examining the source code of `printf`. For instance, **"%"** $\in \Sigma$ has **Pre** = TRUE, and **Post** captures the following semantics: if `printf` is in printing mode (determined by a variable MODE in $\mathcal{V}$), then FMTPTR is incremented, and `printf` enters argument-capture mode. If `printf` is in argument-capture mode, then FMTPTR and DONE are incremented, and `printf` enters printing mode (this corresponds to printing a **"%"** to `stdout`). Formally, $[(\text{MODE} = \text{printing}) \rightarrow (\text{FMTPTR}' = \text{FMTPTR} + 1) \wedge (\text{MODE}' = \text{argument-capture})] \wedge [(\text{MODE} = \text{argument-capture}) \rightarrow (\text{FMTPTR}' = \text{FMTPTR} + 1) \wedge (\text{DONE}' = \text{DONE} + 1) \wedge (\text{MODE}' = \text{printing})]$, where primed variables denote next-state values of the corresponding variables.
- $\mathcal{L}$ is defined to be the language of all possible format-strings, which turns out to be a regular language. We extracted an API-automaton that recognizes all legal format-strings from the control-flow graph of the implementation of `printf`. A portion of this API-automaton is shown in Figure 11 in Appendix A.1.

Several values are possible for **Bad**, and each value determines an attack that exploits format-string vulnerabilities. We present a few values of **Bad** in Section 4.3. In general, this predicate can be expressed as a formula on the elements of $\mathcal{V}$ in a decidable logic that includes quantifier-free Presburger arithmetic, uninterpreted functions, and a theory of memories (arrays). (A formula in quantifier-free Presburger arithmetic consists of a set of linear constraints over integer variables combined using the Boolean operators $\neg$, $\wedge$, and $\vee$.)

We constructed a tool to examine the above system and detect format-string attacks. The tool encodes the

[2]In the actual implementation of `printf`, the flags are C integer and pointer data types, i.e., finite-precision bit-vectors. In our model, flags that just take two values, 0 and 1, are treated as Boolean, while the rest are treated as (unbounded) integers. While this approach achieves efficiency by raising the level of abstraction, it does not model integer overflow, and hence might lead to both false positives and false negatives.

(A) **Bad** for Section 4.3.1	(B) **Bad** for Section 4.3.2
$[\text{FMTPTR} < \text{DIS} + (\text{LEN} - 1) - 1]$ $\wedge\ [\text{ARGPTR} > \text{DIS}]$ $\wedge\ [\text{ARGPTR} < \text{DIS} + (\text{LEN} - 1) - 4]$ $\wedge\ [*\text{FMTPTR} = \text{'\%'}]$ $\wedge\ [*(\text{FMTPTR} + 1) = \text{'s'}]$ $\wedge\ [*\text{ARGPTR} = a_1]$ $\wedge\ [*(\text{ARGPTR} + 1) = a_2]$ $\wedge\ [*(\text{ARGPTR} + 2) = a_3]$ $\wedge\ [*(\text{ARGPTR} + 3) = a_4]$ $\wedge\ [\text{MODE} = \text{printing}]$	$[\text{FMTPTR} < \text{DIS} + (\text{LEN} - 1) - 1]$ $\wedge\ [\text{ARGPTR} > \text{DIS}]$ $\wedge\ [\text{ARGPTR} < \text{DIS} + (\text{LEN} - 1) - 4]$ $\wedge\ [*\text{FMTPTR} = \text{'\%'}]$ $\wedge\ [*(\text{FMTPTR} + 1) = \text{'n'}]$ $\wedge\ [*\text{ARGPTR} = a_1]$ $\wedge\ [*(\text{ARGPTR} + 1) = a_2]$ $\wedge\ [*(\text{ARGPTR} + 2) = a_3]$ $\wedge\ [*(\text{ARGPTR} + 3) = a_4]$ $\wedge\ [\text{DONE} = \text{ATTACK}]$ $\wedge\ [\text{MODE} = \text{printing}]$

Figure 5: Values of the predicate `Bad` used in Section 4.3.1 and Section 4.3.2.

`printf` system, and is parameterized by the values of DIS, LEN, and the predicate **`Bad`**. Our choice of a bounded model checker was mainly influenced by the logic needed to express our model of `printf`, as elaborated below:

1. We need to model certain values in the stack precisely. In particular, we need to track the contents of the format-string because it serves as a concrete counter-example if **`Bad`** is satisfied. This necessitates the use of a theory of memories and uninterpreted functions.
2. `printf` uses both integer and Boolean variables, where the integer variables are modified using linear-arithmetic operations (addition and multiplication by a constant). Thus, to express formulas over these variables, we need quantifier-free Presburger arithmetic.

Based on these requirements, we chose to use a bounded model checker called UCLID [49]. The details of how UCLID works are outside the scope of this paper, and may be found elsewhere [10, 44]; in Appendix A.2, we include a brief description of the subset of UCLID's logic that we make use of. The `printf` system $\mathcal{S}$ can be encoded as an UCLID model in a straightforward manner. If **`Bad`** is satisfied, then UCLID produces a counter-example that can be directly translated to a format-string that demonstrates the attack. At each call-site to `printf`, we only need to examine format-strings of length less than or equal to LEN-1 (we exclude the terminating `'\0'`). Hence, a bound of LEN-1 suffices to make bounded model checking *complete* at that call-site; i.e., a `printf` location deemed safe using our tool with the bound LEN-1 will indeed be safe with respect to the property checked.

4.3 Identifying Attacks

In attacks that we consider, the goal of the attacker is to manipulate the contents of the format-string so as to force ARGPTR to move into the format-string. Hence, ARGPTR has to move by at least DIS bytes by the time FMTPTR moves LEN-1 bytes. Because the attacker controls the value of the format-string, he can control the value of the arguments that `printf` reads from the stack. As demonstrated below, this vulnerability can be used to read data from, or write data to nearly any location in memory.

4.3.1 Reading from an arbitrary location

One of the ways an attacker can print the contents of memory at address $a_4a_3a_2a_1$,[3] where a_4 is the most-significant byte, is to construct a format-string that moves FMTPTR and ARGPTR such that when `printf` is in printing mode and FMTPTR points to the beginning of a `"%s"`, ARGPTR points to the beginning of a sequence of 4 bytes, whose value as a pointer is $a_4a_3a_2a_1$. Then, when `printf` reads a `"%s"`, it interprets the argument at ARGPTR as a pointer and prints the contents of the memory location specified by the pointer as a string, which would let the attacker achieve his goal. This is formalized using the predicate **`Bad`** shown in Figure 5(A). The following features of **`Bad`** are noteworthy:

1. The little-endianness of the machine is reflected in the formulation of **`Bad`**: bytes are arranged from most-significant to least-significant as addresses decrease; thus, for example, a_1 appears at a lower address than a_4.
2. Symbolic values of different stack locations, such as those at FMTPTR and ARGPTR, appear in **`Bad`**, and demon-

[3] a_1, a_2, a_3, a_4 are required to be non-zero, because a zero value is interpreted as `'\0'`, and terminates the format-string.

Sl.no.	DIS	LEN	Read attack (Section 4.3.1)		Write attack (Section 4.3.2)	
			Attack string discovered	Time (sec.)	Attack string discovered	Time (sec.)
(1)	0	7	"$a_1a_2a_3a_4$`%s`"	0.2	No attack possible.	0.3
(2)	4	7	No attack possible.	0.3	No attack possible.	0.3
(3)	4	16	"$a_1a_2a_3a_4$`%d%s`"	0.4	"`%234Lg%n`$a_1a_2a_3a_4$"	4.8
(4)	4	16	"`%Lx%ld%s`$a_1a_2a_3a_4$"	1.0	"$a_1a_2a_3a_4$`%%%229X%n`"	13.1
(5)	8	16	"$a_1a_2a_3a_4$`%Lx%s`"	0.9	"$a_1a_2a_3a_4$`%230g%n`"	22.2
(6)	16	16	"`%Lg%Lg%s`$a_1a_2a_3a_4$"	1.1	"$a_1a_2a_3a_4$`%137g%93g%n`"	106.5
(7)	20	20	"$a_1a_2a_3a_4$`%Lg%g%s`"	5.3	"$a_1a_2a_3a_4$`%210Lg%20g%n`"	148.7
(8)	24	20	"$a_1a_2a_3a_4$`%Lg%Lg%s`"	2.1	"$a_1a_2a_3a_4$`%61Lg%169Lg%n`"	204.2
(9)	32	24	"$a_1a_2a_3a_4$`%g%Lg%Lg%s`"	13.5	"$a_1a_2a_3a_4$`%78Lg%80g%72Lg%n`"	343.5

Figure 6: Some attack patterns generated by our tool. For the write attack, we chose to write the integer 234 to the memory location with address $a_4a_3a_2a_1$.

strate the need to track stack contents precisely.

Figure 6 shows some results produced by the tool for various values of DIS and LEN. For instance, line (3) shows that the format-string "$a_1a_2a_3a_4$`%d%s`" can be used to read the contents of memory at $a_4a_3a_2a_1$ when DIS and LEN are 4 and 16, respectively. The attack proceeds as follows: initially FMTPTR points to the format-string, and ARGPTR is 4 smaller than FMTPTR. `printf` starts execution in printing mode; it advances FMTPTR and prints the bytes a_1, a_2, a_3, and a_4 to `stdout`. When `printf` reads the `'%'`, it advances FMTPTR by one and enters argument-capture mode. When it reads `'d'`, it advances FMTPTR by one, reads an integer (4 bytes) from the location pointed to by ARGPTR, prints this integer to `stdout`, and returns to printing mode. As a result ARGPTR points to the beginning of the format-string, and FMTPTR is positioned at the beginning of the sequence `"%s"`. When `printf` processes the `"%s"`, the contents of memory at location $a_4a_3a_2a_1$ are printed to `stdout`. Other interesting features to note in Figure 6 are:

1. In line (2), the tool is able to infer that an attack is not possible. Intuitively, this is because the format-string is too small to contain a sequence of commands that carry out the desired attack.
2. Lines (3) and (4) present two format-strings for the same parameters. We achieved this by first observing case (3), and running the tool again, appending a suitable term to **`Bad`** to exclude case (3). This technique can be iterated to infer as many variants of this attack as desired. Such attack variants could be used to design input validators that disallow malicious format-strings.

4.3.2 Writing to an arbitrary location

Another kind of format-string attack allows an attacker to write a value of his choice at a location in memory chosen by him. To do so, he makes use of the `"%n"` feature provided by `printf`. When `printf` is in printing mode and encounters a `"%n"` in the format-string, it reads an argument off the stack, which it interprets to be a pointer to an integer. It then writes the value of the flag DONE to this location, where DONE counts the number of bytes that should have been output by `printf`. Figure 5(B) shows the case where an attacker writes the integer ATTACK to the address $a_4a_3a_2a_1$.

Figure 6 shows some format-strings obtained by the tool to write the integer 234 to the address $a_4a_3a_2a_1$ in memory. Consider line (5) for instance; for the values 8 and 16 for DIS and LEN respectively, the tool inferred the format-string "$a_1a_2a_3a_4$`%230g%n`". When `printf` starts execution, it is in printing mode, and ARGPTR is 8 bytes below FMTPTR on the stack. As FMTPTR moves along the format-string, a_1, a_2, a_3, and a_4 (4 bytes) are printed to `stdout`, thus incrementing DONE by 4. The next byte `"%"` increments FMTPTR by 1 byte and forces `printf` into argument-capture mode. The next 3 bytes, `'2'`, `'3'` and `'0'` are treated as width parameters, and `printf` stores the value 230 in an internal flag WIDTH (part of $\mathcal{V}$ for `printf`). When `printf` processes the next byte, `'g'`, it advances ARGPTR by 8 bytes, reads a `double` value from the stack, prints this value (appropriately formatted) to `stdout`, adds the value of WIDTH to DONE, and returns to printing mode. At this point, ARGPTR points to the beginning of the format-string, whose first four bytes contain $a_1a_2a_3a_4$, DONE is 234, and FMTPTR points to the beginning of the sequence `"%n"`. When `printf` processes `"%n"`, the value of DONE is written to

$a_4a_3a_2a_1$, completing the attack.

The times shown in Figure 6 were obtained on a machine with an Intel Pentium-4 processor running at 2GHz, with 1GB of RAM, running Redhat Linux-7.2. All runs completed within a few minutes. As a general trend, the time taken increases as LEN increases, though not monotonically. The reason is that for larger values of LEN, it is required to run the bounded model checker UCLID for more steps, leading to a larger formula for it to check. UCLID translates the problem into one of checking the validity of a Boolean formula, which is checked using a SAT solver called Siege [46]. Note also that the time taken for finding read attacks is much lower than that for finding write attacks. This is because finding a write attack involves solving a more constrained problem than for the read attack: In addition to finding a sequence of conversion specifications that moves ARGPTR into the format-string, one needs to find associated width values that add up to the desired value (234 in Figure 6). Furthermore, this sequence of API operations must fit within the format-string.

The write attack results in Figure 6 are for writing a specific value to a specific address. One can also ask whether it is possible to write any value to any address, a conservative test which can be performed faster since the problem is less constrained. For example, for line (9) of Figure 6, we were able to perform this check in 168 seconds, about half the time needed to perform the precise check.

4.4 Optimizations

In our model of the `printf` system, *each byte* in the format-string is considered as an API operation. An alternative technique would be to use "summary" operations infered by the tool as building blocks to generate format-strings. For instance, the tool automatically discovered that if `printf` is in printing mode, then the sequence of API operations `"%Lg"` moves FMTPTR by 3 bytes, ARGPTR by 12 bytes, and reads a `long double` value. Such sequences of API operations could be used as building blocks and composed together to discover format-string vulnerabilities. A tool based on such techniques would potentially work faster than the basic technique presented.

5 The IBM CCA API

The IBM CCA API [29] is a cryptographic key-management API that is used with hardware security modules (secure coprocessors) such as the IBM 4758. The IBM 4758 is widely used in commerce and banking, and has received the highest possible rating for its physical security (FIPS level-4). This section focuses on a vulnerability in the CCA API that was first discovered by Bond [7, 28]. We first provide some background information on the IBM 4758 and the CCA API, and use the technique developed in Section 3 to discover the vulnerability automatically.

5.1 Background on CCA

The IBM 4758 is a coprocessor, and is used together with a host computer. Every IBM 4758 is associated with a distinct *master key*; we will use the notation MK to denote master keys. Each coprocessor safeguards its master key; physical security ensures that MK cannot be retrieved by physically attacking the device. The security of a system built using the IBM 4758 is bootstrapped by safeguarding a single secret, namely MK.

CCA is often used together with the IBM 4758 for key-management tasks (although the IBM 4758 can be used without CCA as well). An important requirement of any key-management service is that the clear values of keys should never be revealed. CCA manages keys by storing them encrypted under the master key MK of the IBM 4758 it operates with. Before we develop this point further, we must introduce the concept of a *control vector* [32, 33], whose design and implementation is an important contribution of CCA.

Every key managed by CCA is associated with a control vector. Control vectors are used by CCA to implement role-based access control (RBAC) [24]. The value of the control vector associated with a key decides the subset of operations from the API that the key has access to. For instance, CCA defines control vectors for PIN keys, data keys, keys for message authentication codes, and so on. The values of control vectors are published by IBM and are publicly available.

Because it is important to maintain the integrity of RBAC, each key and its control vector should be tightly coupled, and any attempt to change the value of the control vector should render the key unusable. There are several options available to implement this, and CCA implements this by using the value of the control vector associated with a key to encrypt it [32, 33]. As mentioned earlier, a key is also stored encrypted under the master key MK

MK	:	Master key of the coprocessor CCA operates with.
KEK	:	Clear value of the key-encrypting key.
K	:	Clear value of a CCA key.
$\mathsf{CV_K}$	:	Control vector associated with K.
$\mathsf{CV_{KEK}}$	:	Control vector for key-encrypting keys.

API command	Expected Input 1	Expected Input 2	Output
Key_Part_Import	KP_1 (clear)	KP_2 (clear)	$(E_{\mathsf{MK}\oplus\mathsf{CV_{KEK}}}(\mathsf{KP}_1\oplus\mathsf{KP}_2), \mathsf{CV_{KEK}})$
Key_Export	$(E_{\mathsf{MK}\oplus\mathsf{CV_{KEK}}}(\mathsf{KEK}), \mathsf{CV_{KEK}})$	$(E_{\mathsf{MK}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$	$(E_{\mathsf{KEK}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$
Key_Import	$(E_{\mathsf{MK}\oplus\mathsf{CV_{KEK}}}(\mathsf{KEK}), \mathsf{CV_{KEK}})$	$(E_{\mathsf{KEK}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$	$(E_{\mathsf{MK}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$

Figure 7: Some commands from the IBM CCA API.

of the IBM 4758 that the CCA operates with. The CCA achieves both these requirements by encrypting key K as $E_{\mathsf{MK}\oplus\mathsf{CV_K}}(\mathsf{K})$ where $\mathsf{CV_K}$ is the control vector associated with K.[4] A symmetric key algorithm, such as 3DES, is used to perform encryption. Each key is stored on the hard disk of the host computer (that the IBM 4758 operates with) as an *operational key-token*, which contain several pieces of information related to the key. For the purpose of this paper, an operational key-token will refer to just two of the components and will be denoted as $(E_{\mathsf{MK}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$. That is, it contains the encrypted value of K, and the clear value of $\mathsf{CV_K}$. When presented with the key-token, the IBM 4758 can use $\mathsf{CV_K}$ from the key-token to compute $\mathsf{MK}\oplus\mathsf{CV_K}$. This value is used to decrypt $E_{\mathsf{MK}\oplus\mathsf{CV_K}}(\mathsf{K})$ and retrieve K within the coprocessor. Of course, this clear value should not be revealed outside the IBM 4758. This operational key-token will not function with another IBM 4758 because the master keys will be different.

We now consider communication between two parties, A and B, each of which has an IBM 4758 (with master keys $\mathsf{MK_A}$ and $\mathsf{MK_B}$, respectively) and uses the CCA API for key management (each party has a host computer and a coprocessor). One of the supported methods for communication involves establishing a communication channel between A and B, and setting up a symmetric *key-encrypting key* to encrypt all CCA-managed keys that are transported over the channel in further communications. The key-encrypting key, whose clear value we denote by KEK, is itself stored as a CCA key, and is associated with the control vector for key-encrypting keys: $\mathsf{CV_{KEK}}$. It is stored at A and B as an operational key-token with values $(E_{\mathsf{MK_A}\oplus\mathsf{CV_{KEK}}}(\mathsf{KEK}), \mathsf{CV_{KEK}})$ and $(E_{\mathsf{MK_B}\oplus\mathsf{CV_{KEK}}}(\mathsf{KEK}), \mathsf{CV_{KEK}})$, respectively. One of the methods supported by CCA for installing key-encrypting keys works as follows: One of the parties, say A, generates two (or more) *key parts*, KP_1 and KP_2, such that $\mathsf{KEK} = \mathsf{KP}_1 \oplus \mathsf{KP}_2$. These key parts are transported (in the clear) separately to B, where they are entered using the CCA command Key_Part_Import (see Figure 7). The result of this command is an operational key-token for KEK. The idea is that the clear value of KEK cannot be retrieved unless all the key-part holders collude.[5]

Now suppose that A has a key K associated with control vector $\mathsf{CV_K}$, stored at A as $(E_{\mathsf{MK_A}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$, that it wants to share with B. Clearly, this key-token cannot be used by B because the clear value of K is encrypted with $\mathsf{MK_A}$. Hence, to make the key-token "device-independent", CCA provides an API command Key_Export, which is specified in Figure 7. This command takes as input the operational key-tokens corresponding to KEK and K and produces the value $(E_{\mathsf{KEK}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$. This value is device-independent, and is called an *export key-token*. Intuitively, the key-token $(E_{\mathsf{MK_A}\oplus\mathsf{CV_{KEK}}}(\mathsf{KEK}), \mathsf{CV_{KEK}})$ is used to retrieve the value of KEK, which is then used to produce $\mathsf{KEK}\oplus\mathsf{CV_K}$, where $\mathsf{CV_K}$ is retrieved from the key-token $(E_{\mathsf{MK_A}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$. The IBM 4758 can also retrieve the value K from $(E_{\mathsf{MK_A}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$. These values can then be used to produce the export key-token.

The export key-token can be transported over the network to B, where it is referred to as an *import key-token*. At B, an API command Key_Import is used to convert this key-token into an operational key-token for B. The details of this command are shown in Figure 7. The first input to this command is the operational key-token of KEK, while the second input is the value of the key-token received over the communication channel. As with Key_Export, Key_Import first retrieves the clear value of KEK, and uses this value with the value of $\mathsf{CV_K}$ from the second input to produce $\mathsf{KEK}\oplus\mathsf{CV_K}$. This value is used to retrieve K by decrypting $E_{\mathsf{KEK}\oplus\mathsf{CV_K}}(\mathsf{K})$. The clear value of K and the value of $\mathsf{CV_K}$ are then used to produce an operational key-token $(E_{\mathsf{MK_B}\oplus\mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$, which can be used at B.

[4] $\oplus$ denotes bit-wise exclusive-or; $E_{\mathsf{K}}(\mathsf{P})$ and $D_{\mathsf{K}}(\mathsf{P})$ denote encryption and decryption of P using key K, respectively.

[5] This technique is insecure, and IBM [28] recommends the use of public-key encryption to transport key parts. However, this mode of key-part transportation continues to be supported.

5.2 Formalizing the API

We will now formalize the CCA API using the framework developed in Section 3. Our focus is on the security of the CCA API, and hence we will restrict our attention to the sequence of API operations that can be issued on just *one* coprocessor. We make the following assumptions:

1. A and B communicate, and normal operation proceeds as described in Section 5.1. In particular, A has a key K associated with control vector $\mathsf{CV_K}$ that it wishes to share with B.
2. A sends the key-token ($E_{\mathsf{KEK}\oplus\mathsf{CV_K}}(\mathsf{K})$, $\mathsf{CV_K}$) over the communication channel to B. Here A assumes that the value of the shared key-encrypting key is KEK; that is, the key-token at A corresponding to the shared key is ($E_{\mathsf{MK_A}\oplus\mathsf{CV_{KEK}}}(\mathsf{KEK})$, $\mathsf{CV_{KEK}}$).
3. The key-encrypting key is installed at B using the Key_Part_Import command, and the two key parts are KP_1 and KP_2, where KEK = $\mathsf{KP}_1 \oplus \mathsf{KP}_2$. Moreover, we assume that the attacker knows *one* of the key parts, say KP_2, but not the other key part. The attacker is not assumed to collude with the holder of KP_1.
4. The attacker has complete control over B. In particular, the attacker can (a) manipulate any value sent to B over the communication channel, (b) manipulate any key-token stored on the host computer at B, and (c) issue any CCA API command to the coprocessor at B with inputs of his choice. These are standard assumptions following the Dolev-Yao attacker model [22].

Informally, the safety property that we will attempt to verify is that every operational key-token obtained at B using Key_Import should be associated with the same control vector as the control vector associated with the export key-token sent by A. That is, if the value sent by A over the communication channel is ($E_{\mathsf{KEK}\oplus\mathsf{CV_K}}(\mathsf{K})$, $\mathsf{CV_K}$), then the operational key-token that must result at B is ($E_{\mathsf{MK_B}\oplus\mathsf{CV_K}}(\mathsf{K})$, $\mathsf{CV_K}$).

Using the framework developed in Section 3, we have $\mathcal{S}$ = ($\mathcal{V}$, **Init**, Σ, $\mathcal{L}$) where,

- $\mathcal{V}$ denotes a single set-valued variable $keytokens$, which denotes the set of all key-tokens known to B.
- **Init**: $keytokens = \emptyset$, the empty set.
- Σ = {Key_Part_Import, Key_Import}
- $\mathcal{L} = \Sigma^*$

The predicate **Bad** is defined as ($E_{\mathsf{MK_B}\oplus\mathsf{CV_{new}}}(\mathsf{K})$, $\mathsf{CV_{new}}$) $\in keytokens$, where $\mathsf{CV_{new}} \neq \mathsf{CV_K}$. Intuitively, we keep track of the set of key-tokens available on the IBM 4758 using the variable $keytokens$, and assume that this set is initially empty. We will only model two operations from the CCA API, namely Key_Import and Key_Part_Import, and assume that these can be interleaved in any order, denoted by $\mathcal{L} = \Sigma^*$. The operations in Σ accept two arguments each, and **Pre** and **Post** are defined as follows:

1. Key_Part_Import(arg_1, arg_2): **Pre**(arg_1, arg_2) is TRUE and **Post**(arg_1, arg_2) asserts that ($E_{\mathsf{MK_B}\oplus\mathsf{CV_{KEK}}}(\mathsf{arg}_1\oplus\mathsf{arg}_2)$, $\mathsf{CV_{KEK}}$) $\in keytokens$.
2. Key_Import(arg_1, arg_2): **Pre**(arg_1, arg_2) asserts that arg_1 and arg_2 have the structure of key-tokens. Let $\mathsf{arg}_1^{\mathsf{enc}}$ and $\mathsf{arg}_1^{\mathsf{cv}}$ denote the first and second half respectively of the key-token arg_1, and similarly for arg_2. **Post**(arg_1, arg_2) asserts that $E_{\mathsf{MK_B}\oplus\mathsf{arg}_2^{\mathsf{cv}}}(\mathsf{Key}) \in keytokens$, where we define $\mathsf{Key} = D_{\mathsf{Val}\oplus\mathsf{arg}_2^{\mathsf{cv}}}(\mathsf{arg}_2^{\mathsf{enc}})$, and $\mathsf{Val} = D_{\mathsf{MK_B}\oplus\mathsf{arg}_1^{\mathsf{cv}}}(\mathsf{arg}_1^{\mathsf{enc}})$.

$$
\begin{array}{rcll}
\oplus \text{ rules} & : & \dfrac{\Gamma \vdash a \quad \Gamma \vdash b}{\Gamma \vdash a \oplus b} & \dfrac{\Gamma \vdash a\oplus b \quad \Gamma \vdash b}{\Gamma \vdash a} \\[2ex]
\text{(En/De)cryption} & : & \dfrac{\Gamma \vdash k \quad \Gamma \vdash p}{\Gamma \vdash E_k(p)} & \dfrac{\Gamma \vdash k \quad \Gamma \vdash E_k(p)}{\Gamma \vdash p} \\[2ex]
\text{(Un)pairing} & : & \dfrac{\Gamma \vdash a \quad \Gamma \vdash b}{\Gamma \vdash (a,b)} & \dfrac{\Gamma \vdash (a,b)}{\Gamma \vdash a \quad \Gamma \vdash b}
\end{array}
$$

Figure 8: Knowledge enhancement by the attacker.

Intuitively, Val denotes the clear value of the key-encrypting key, retrieved from arg_1, and this value is used to retrieve the value Key from arg_2. This value is then used to produce an operational key-token, which is required by **Post** to be in $keytokens$.

In accordance with our assumptions, the attacker can use any value that he knows as an argument to the API operations. The attacker initially knows (a) ($E_{\mathsf{KEK}\oplus\mathsf{CV_K}}(\mathsf{K})$, $\mathsf{CV_K}$), which he can learn from the communication channel, (b) KP_2, namely, the clear value of one of the key parts that is used to construct KEK, and (c) values of control vectors, which are publicly known. The attacker can apply rules such as those shown in Figure 8 to enhance his knowledge. In the figure, Γ is used to denote the set of terms known to the attacker, and the rules capture how the attacker can enhance his knowledge using the set of terms that he knows. For instance, the first rule says that if the attacker knows two terms a and b, he can combine these to know $a \oplus b$. These rules correspond to the rules of the Dolev-Yao model augmented with the capability to manipulate terms by combining them with exclusive-or. Standard rules such as commutativity and associativity apply for terms generated using exclusive-or, and we have omitted these from Figure 8.

When the above API is presented to our model checker, which is described in detail in Section 5.3, we obtain the counter-example trace shown in Figure 9. This counter-example denotes the "chosen-difference attack" on control vectors, first discovered manually by Bond [7].

In statement (1) of Figure 9, the attacker installs a key of his choice as the key-encrypting key at B. Recall that the attacker knows KP_2, where $\mathsf{KP}_1 \oplus \mathsf{KP}_2 = \mathsf{KEK}$, and KP_1 is not known to the attacker. However, the attacker can manipulate key part KP_2 using the rules presented in Figure 8; he modifies KP_2 to $\mathsf{KP}_2 \oplus \mathsf{CV}_\mathsf{K} \oplus \mathsf{CV}_\mathsf{new}$, where CV_K is the control vector of the key transported over the network, and CV_new is another control vector, chosen by the attacker. When $\mathsf{Key\_Part\_Import}$ is executed with the modified key part as the second argument, the key-token $(\mathsf{E}_{\mathsf{MK_B} \oplus \mathsf{CV_{KEK}}}(\mathsf{KEK} \oplus \mathsf{CV_K} \oplus \mathsf{CV_{new}}), \mathsf{CV_{KEK}})$ results, and B thinks that this is the key-token for the shared key-encrypting key.

(1) $\mathsf{Key\_Part\_Import}$:
Input 1: KP_1
Input 2: $\mathsf{KP}_2 \oplus \mathsf{CV_K} \oplus \mathsf{CV_{new}}$
Output: $(\mathsf{E}_{\mathsf{MK_B} \oplus \mathsf{CV_{KEK}}}(\mathsf{KEK} \oplus \mathsf{CV_K} \oplus \mathsf{CV_{new}}), \mathsf{CV_{KEK}})$
(2) $\mathsf{Key\_Import}$:
Input 1: $(\mathsf{E}_{\mathsf{MK_B} \oplus \mathsf{CV_{KEK}}}(\mathsf{KEK} \oplus \mathsf{CV_K} \oplus \mathsf{CV_{new}}), \mathsf{CV_{KEK}})$
Input 2: $(\mathsf{E}_{\mathsf{KEK} \oplus \mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_{new}})$
Output: $(\mathsf{E}_{\mathsf{MK_B} \oplus \mathsf{CV_{new}}}(\mathsf{K}), \mathsf{CV_{new}})$

Figure 9: Counter-example trace exposing vulnerability.

Input 2 of Statement (2) of Figure 9 corresponds to a step in which the attacker first uses the unpairing and pairing rules in Figure 8 to obtain $(\mathsf{E}_{\mathsf{KEK} \oplus \mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_{new}})$ from $(\mathsf{E}_{\mathsf{KEK} \oplus \mathsf{CV_K}}(\mathsf{K}), \mathsf{CV_K})$, a value that he knows. Second, he invokes $\mathsf{Key\_Import}$ with this modified key-token and the key-token of the shared key obtained in the first step of the attack. As explained earlier, $\mathsf{Key\_Import}$ produces $\mathsf{MK_B} \oplus \mathsf{CV_{KEK}}$ using the value of $\mathsf{CV_{KEK}}$ from Input 1, which is then used to retrieve $\mathsf{KEK} \oplus \mathsf{CV_K} \oplus \mathsf{CV_{new}}$ from the first half of Input 1. Note that under normal operation this would have retrieved the value KEK instead. $\mathsf{Key\_Import}$ then extracts $\mathsf{CV_{new}}$ from Input 2, and xor's this with $\mathsf{KEK} \oplus \mathsf{CV_K} \oplus \mathsf{CV_{new}}$ to obtain $\mathsf{KEK} \oplus \mathsf{CV_K}$. This value is used to retrieve K from the portion $\mathsf{E}_{\mathsf{KEK} \oplus \mathsf{CV_K}}(\mathsf{K})$ of Input 2. Note that in the process, B has been fooled into thinking that the key is associated with the control vector $\mathsf{CV_{new}}$. Hence, $\mathsf{Key\_Import}$ terminates by producing an operational key-token $(\mathsf{E}_{\mathsf{MK_B} \oplus \mathsf{CV_{new}}}(\mathsf{K}), \mathsf{CV_{new}})$; this violates the safety criterion, which demands that operational key-tokens should be associated with the same control vector at B as was intended by A. As explained earlier, violating the safety condition compromises the integrity of RBAC; Bond [7] demonstrates how this can be used to learn sensitive values, such as PIN-encrypting keys.

5.3 The Model Checker

We now discuss the implementation of the bounded model checker used to discover the attack presented in Figure 9. A formal description of the API as presented in Section 5.2 is presented to the bounded model checker. Informally, the bounded model checker "unwinds" the API-automaton for $\mathcal{L}$ and produces all paths of length less than or equal to N, where N is the bound for the model checker. Each reachable state is then checked to see if the safety property in question, namely $\neg$**`Bad`**, is satisfied.

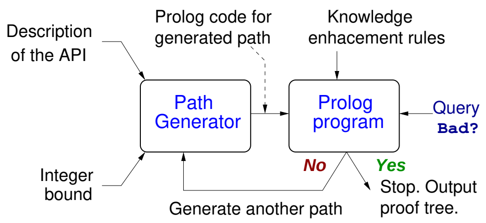

Figure 10: Design of the bounded model checker.

A schematic diagram of the bounded model checker is shown in Figure 10. The model checker uses Prolog as its underlying decision procedure. The choice of Prolog was motivated by two requirements: (a) we needed to encode inference rules, such as those presented in Figure 8, to model knowledge enhancement by the attacker, and (b) we needed to produce concrete values for the variables involved in the above rules in case a counter-example was found. Prolog satisfies both these requirements; its capability to perform unification provides us with a concrete counter-example trace if one exists. Moreover, it is also easy to encode transformations corresponding to API operations as Prolog statements.

The main component of the model checker is a path generator that accepts the API-automaton for $\mathcal{L}$ and an integer bound N. It exhaustively generates all paths of length up to N, where each path corresponds to a sequence of API operations. Each path also corresponds to a Prolog program obtained by combining the Prolog statements of the API operations in the path with initial knowledge of the attacker and the knowledge-enhancing rules shown in Figure 8.

In our implementation, we simplified the first rule presented in Figure 8 by assuming that only ground terms (i.e., terms without the encryption symbol) could be combined using exclusive-or. This simplification restricts the set of terms that can be learned by the attacker, but was sufficient to detect the attack. The general model with Dolev-Yao rules but allowing arbitrary terms to be combined with exclusive-or was recently studied [14, 18], and the insecurity decision problem was shown to be decidable. Hence, it must be emphasized that it is possible to solve the problem with the rules as presented in Figure 8, and that our simplification was solely an implementation consideration.

To discover the attack presented in Figure 9, a bound of $N = 2$ suffices. When the model checker explores the path Key_Part_Import $\rightarrow$ Key_Import of the API-automaton, the Prolog statements corresponding to these API operations are emitted, and combined with the Prolog statements for the rules in Figure 8. The resulting Prolog program satisfies the predicate **`Bad`**, and the proof tree generated by Prolog provides concrete values to the arguments of the API operations in the path explored. This results in the trace shown in Figure 9.

IBM recommends the use of procedural controls to avoid the attack discussed above [28]. One such procedural control is the use of *verification patterns*, which are akin to hash functions. A generates a verification pattern $\mathsf{VP_A}$ for KEK, and transports it separately to B. When the key-encrypting key is installed using Key_Part_Import at B, a verification pattern $\mathsf{VP_B}$ is generated and compared with $\mathsf{VP_A}$. If the verification patterns do not match, it means that the key parts have been tampered with; the resulting key is unsafe, and must not be used. In a second experiment, we modified the semantics of Key_Part_Import, so that it compared the verification patterns of the combined key parts and KEK. With this modification, we were no longer able to discover the attack shown in Figure 9.

6 Discussion

We consider several applications of our work. The technique can be used to analyze APIs, and the resulting error traces can be used by static checkers, such as MOPS and meta-compilers, and by runtime checkers, such as reference monitors. They can also be used as patterns in signature-based misuse-detection systems, such as Snort [40].

If the set of traces of allowed API operations $\mathcal{L}$ for a system $\mathcal{S}$ is finite, and arguments to the API operations are from a finite domain, then the technique can be used to produce all possible traces of API operations that violate a safety property. This is achieved by iteratively running the model checker, cumulatively excluding paths corresponding to counter-examples. These traces can then be used directly with the checkers mentioned above.

If $\mathcal{L}$ is infinite or the API operations accept inputs from an infinite domain, then it is not possible to enumerate all possible traces. In such cases, an option would be to generate a number of counter-examples and use learning techniques on the traces to identify a class of vulnerabilities. For instance, Chen *et al.* argue that it is potentially dangerous to call execl() after calling seteuid(0), but before dropping privileges [12]. A tool based on our technique that analyzes system calls as API operations, with an appropriate safety property, would possibly discover this sequence. It would also discover sequences that contain non-user-id-setting system calls between seteuid(0) and execl(), because those sequences would also have the same semantics with respect to the property at hand. The problem of learning patterns from a finite set of malicious traces has also been addressed in research on anomaly detection [43], and those techniques could be brought to bear in this situation, as well.

Our work can also be used for API-level test-vector generation. Traces of API operations produced by our technique can be used to test revised versions of the API, as we did in Section 5.3. Recent work has shown that model checking is useful for test-vector generation [1, 5].

To apply our technique to analyze the API of a system, a formal specification of the system as described in Section 3 is required. Formal specifications of the system and the semantics of API operations may not always be available, in which case an option is to extract them from an implementation. In the case of software systems, this may be possible by analyzing source code, as we did for the case of `printf`. In other systems, and in cases where source code is not available, simulation may be used. For instance, Chen *et al.* extract the effect of UNIX user-id-setting system calls, such as setuid, using a simulator that tries the system call from every possible state of the system [13].

7 Related Work

Software model checking. There are several model-checking tools that check software for violations of temporal ordering rules on API calls. MOPS [12] and meta-compilation [3, 23] use context-sensitive pattern matching to check source code for the presence of erroneous API-usage patterns. While these tools sacrifice precision of results in the interest of scalability, SLAM [4] achieves precision by doing a more thorough analysis, based on predicate-abstraction [25]. To scale, it automatically abstracts away data-flow details that are irrelevant to verify the property in question. Vault [21] is a C-like programming language that allows users to express domain-specific resource-management protocols, such as ordering restrictions on operations, and uses type checking to enforce these rules. The common theme in all the above projects is to identify erroneous API-usage patterns by examining source code. Ammons *et al.* [2] have worked on using machine learning on runtime execution traces of programs that use API calls to infer protocol rules that APIs must follow. CHIC [11] is a tool that checks compatibility of interactions between software modules. For instance, it can check that API-level assumptions made by one module about other modules are consistent with the guarantees provided by those modules.

Our research is complementary to the projects discussed above—it focuses instead on analyzing *API specifications* for vulnerabilities. We assume that a specification of the API is provided, and analyze it to check if there are sequences of API calls that compromise the security of the system. As discussed in Section 6, such sequences of API operations could potentially benefit the tools discussed above. Additionally, our work requires precisely modeling data values, as was exemplified in our case studies. Prior work [3, 4, 12, 23] has mostly addressed checking client software for control-flow-intensive properties.

Security-protocol verification. The use of model checking in protocol analysis has been explored by several researchers. Lowe [30] used the model checker FDR to find an attack against the Needham-Schroeder protocol. Mitchell *et al.* [35, 36] demonstrated the use of a general-purpose model checker called Murφ to verify several security-protocols, including SSL-3.0. These tools suffer from the state-space-explosion problem and can only analyze a bounded number of protocol sessions. The use of theorem provers has been investigated to analyze an unbounded number of protocol sessions, albeit at the cost of losing complete automation–the NRL Analyzer [34] and Isabelle [39] are two tools that adopt this approach. Both these tools encode the behaviour of honest participants of the protocol and the messages exchanged in the protocol. The NRL Analyzer starts with the description of an insecure state, and uses backtracking and unification to see if the state is reachable. On the other hand, Isabelle uses the description of the protocol rules to inductively describe the set of possible traces that could arise. Athena [47] and Brutus [16] are two special-purpose model checkers for security-protocols that combine state space exploration with theorem proving. While most of these tools assume the Dolev-Yao [22] adversary model, recent work [14, 18] has considered extensions to this model; we considered such an extended model in Section 5.

Our work can be applied to verify a limited class of security protocols; we could use techniques similar to [36] to model protocols with a bounded number of sessions. However, our framework cannot model an unbounded number of sessions. In addition, protocol verifiers explicitly mention the principals involved in the protocol, and often treat the adversary as one of the principals, who can intercept, modify, misdirect, or retransmit messages. Our framework does not accommodate the notion of principals, and our notion of the adversary is rather simple—it assumes that he has complete control over the system, and can apply API operations in any allowed order to compromise the security of the system. It is possible to model knowledge enhancement in some cases—as was demonstrated in Section 5. Security-protocol verifiers can also be used to verify certain classes of API-level vulnerabilities; for instance, the subset of the IBM CCA API that we modeled could have been modeled and analyzed with the NRL Analyzer, as well. However, verifying API-safety for a general system requires a framework that precisely captures the changes that an API operation makes to an individual process's state, as was illustrated in the `printf` case study. Protocol analyzers are geared towards analyzing the messages exchanged, and how these can be used to enhance the adversary's knowledge. We are unaware of protocol verifiers that capture system state in as precise a manner as would be required to check for API-safety.

Other work. Manadhata and Wing [31] address the problem of measuring security, and give a formal definition, and techniques to measure the *attack-surface metric*, using a framework similar to ours. This metric includes the API of the system, and the resources of the system that the API modifies. However, they do not give an algorithm to analyze APIs to identify traces that could violate security properties.

There has also been work specific to the case studies considered in this paper. Shankar *et al.* [45] have used type qualifiers to identify format-strings that could potentially be controlled by the attacker. Their technique adds annotations to variable types and uses program analysis to propagate the annotations to identify "tainted" format-strings. FormatGuard [19] is a `glibc` patch that detects possible exploits against format-string vulnerabilities at runtime. Bond and Anderson [8, 9] have studied the IBM CCA API and several other cryptographic-key-management APIs, and have identified several weaknesses in them. However, they have not developed a general technique to analyze APIs automatically. We believe our technique is a first step towards providing an automatic technique to identify API-level vulnerabilities.

References

[1] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *2nd IEEE International Conference on Formal Engineering Methods (ICFEM)*, December 1998.

[2] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *29th Symposium on Principles of Programming Languages (POPL)*, January 2002.

[3] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, November 2002.

[4] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *29th Symposium on Principles of Programming Languages (POPL)*, January 2002.

[5] D. Beyer, A. J. Chipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *26th International Conference on Software Engineering (ICSE)*, May 2004. To appear.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, March 1999.

[7] M. Bond. A chosen key difference attack on control vectors. *Unpublished Manuscript*, November 2000. http://www.cl.cam.ac.uk/∼mkb23/research/CVDif.pdf (URL circa May 2004).

[8] M. Bond. Attacks on cryptoprocessor transaction sets. In *3rd Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, May 2001.

[9] M. Bond and R. Anderson. API-level attacks on embedded systems. *IEEE Computer*, 34(10):67–75, October 2001.

[10] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *14th International Conference on Computer Aided Verification (CAV)*, July 2002.

[11] A. Chakrabarti, L. de Alfaro, and T. A. Henzinger. Interface compatibility checking for software modules. In *14th International Conference on Computer Aided Verification (CAV)*, July 2002.

[12] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *9th ACM Conference on Computer and Communications Security (CCS)*, November 2002.

[13] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *11th USENIX Security Symposium*, August 2002.

[14] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. An NP decision procedure for protocol insecurity with XOR. In *18th IEEE Symposium on Logic in Computer Science (LICS)*, June 2003.

[15] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, July 2001.

[16] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):443–487, October 2000.

[17] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completness and complexity of bounded model checking. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, January 2004.

[18] H. Comon-Lundh and V. Shmatikov. Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In *18th IEEE Symposium on Logic in Computer Science (LICS)*, June 2003.

[19] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from `printf` format-string vulnerabilities. In *10^{th} USENIX Security Symposium*, August 2001.

[20] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *18^{th} International Conference on Automated Deduction (CADE)*, July 2002.

[21] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[22] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[23] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4^{th} Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[24] D. F. Ferraiolo and D. R. Kuhn. Role based access control. In *15^{th} National Computer Security Conference*, October 1992.

[25] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *9^{th} International Conference on Computer Aided Verification (CAV)*, June 1997.

[26] M. Harrison, W. Ruzzo, and J. Ullmann. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[27] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison Wesley, Boston, MA, February 2004.

[28] IBM Corporation. IBM comment on "A chosen key difference attack on control vectors". *Unpublished Manuscript*, January 2001. http://www.cl.cam.ac.uk/∼mkb23/research/CVDif-Response.pdf (URL circa May 2004).

[29] D. B. Johnson, G. M. Dolan, M. J. Kelly, A. V. Le, and S. M. Matyas. Common cryptographic architecture cryptographic application programming interface. *IBM Systems Journal*, 30(2):130–150, 1991.

[30] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *2^{nd} International Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, March 1996.

[31] P. Manadhata and J. M. Wing. Measuring a system's attack surface. Technical Report CMU-CS-04-102, Computer Science Department, Carnegie Mellon University, January 2004.

[32] S. M. Matyas. Key handling with control vectors. *IBM Systems Journal*, 30(2):151–174, 1991.

[33] S. M. Matyas, A. V. Le, and D. G. Abraham. A key management scheme based on control vectors. *IBM Systems Journal*, 30(2):175–191, 1991.

[34] C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[35] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *IEEE Symposium on Security and Privacy*, May 1997.

[36] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite state analysis of SSL-3.0. In *7^{th} USENIX Security Symposium*, January 1998.

[37] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38^{th} Design Automation Conference (DAC)*, June 2001.

[38] T. Newsham. Format string attacks. September 2000. http://www.securityfocus.com/guest/3342 (URL circa May 2004).

[39] L. Paulson. Proving properties of security protocols by induction. In *IEEE Symposium on Security and Privacy*, May 1997.

[40] M. Roesch. Snort: Lightweight intrusion detection for networks. In *13^{th} Systems Administration Conference (LISA)*, November 1999.

[41] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, February 2000.

[42] S. Schwoon. *Model-Checking pushdown systems*. PhD thesis, Technische Universität München, 2002.

[43] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, May 2001.

[44] S. A. Seshia and R. E. Bryant. Deciding quantifier-free Presburger formulas using parameterized solution bounds. In *19th IEEE Symposium on Logic in Computer Science (LICS)*, July 2004. To appear.

[45] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Automated detection of format-string vulnerabilities using type qualifiers. In *10th USENIX Security Symposium*, August 2001.

[46] Siege SAT solver. http://www.cs.sfu.ca/~loryan/personal (URL circa May 2004).

[47] D. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1,2):47–74, 2001.

[48] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A cooperating validity checker. In *14th International Conference on Computer Aided Verification (CAV)*, July 2002.

[49] UCLID: A verification tool for infinite state systems. http://www.cs.cmu.edu/~uclid (URL circa May 2004).

A Appendix

A.1 API-automaton for $\mathcal{L}$ in `printf`

Figure 11 presents a portion of the API-automaton for `printf`. We do not show the entire API-automaton because it is too large. The API-automaton for `printf` was extracted by examining the control-flow structure of the code in `vfprintf.c` in `glibc-2.3`.

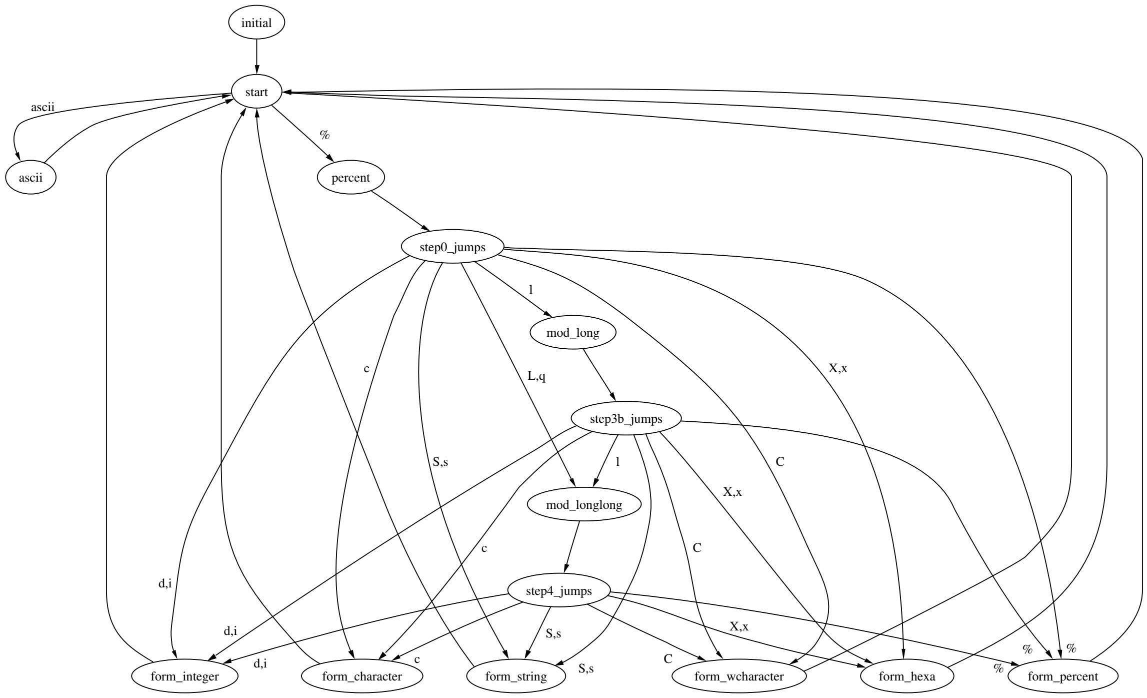

Figure 11: A portion of the API-automaton for `printf`.

A.2 The logic used to model `printf`

Figure 12 summarizes the expression syntax of the logic used to model the `printf` system described in Section 4.2. This logic is a subset of that handled by the bounded model checker UCLID [49]. There are three types of expressions: integer (*int-expr*), Boolean (*bool-expr*), and functions that map integer-valued arguments to an integer value (*function-symbol*).

The simplest Boolean expressions are the values TRUE and FALSE. Boolean expressions are also formed by combining Boolean variables, equalities, or inequalities using Boolean connectives.

Integer expressions are either integer constants or variables, or formed by applying arithmetic functions "+" (addition of integer expressions) and "$*$" (multiplication by an integer constant) to integer arguments, using the *ITE* ("if-then-else") operator, or applying an *uninterpreted function symbol* to a list of integer expressions. The *ITE* operator chooses between two values based on a Boolean control value; i.e., *ITE*(TRUE, x_1, x_2) yields x_1

$$
\begin{aligned}
\textit{bool-expr} \quad ::= \quad & \text{TRUE} \mid \text{FALSE} \mid \textit{bool-var} \mid \neg\textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \\
& \mid (\textit{bool-expr} \vee \textit{bool-expr}) \mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \\
\textit{int-expr} \quad ::= \quad & c \mid \textit{int-var} \mid \textit{function-symbol}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
& \mid \textit{int-expr} + \textit{int-expr} \mid c * \textit{int-expr} \mid \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr})
\end{aligned}
$$

Figure 12: Syntax of the logic used to model `printf`. Expressions can denote computations of Boolean values, integers, or functions yielding integers; c denotes an integer constant.

while *ITE*(FALSE, x_1, x_2) yields x_2. The arithmetic functions and the *ITE* operator enable us to model conditional updates to the flags used by `printf`. Uninterpreted function symbols are used to model functions: nothing is assumed about the meaning of the function except that it maps equal arguments to equal values. They can be used in modeling memories, like the stack in the `printf` system. A memory can be viewed as a function M that maps addresses to data values. Thus, reading from M at address a simply yields the value $M(a)$. (More about how to use UCLID's logic to model memories and arrays can be found in [10].)

The value of a well-formed expression in this logic is defined relative to an interpretation I of the Boolean and integer variables, and the function symbols. Let $\mathbb{Z}$ denote the set of integers. Interpretation I assigns to each variable a Boolean or integer value, and to each function symbol of arity k a function from $\mathbb{Z}^k$ to $\mathbb{Z}$. Given an interpretation I of the variables and function symbols and a well-formed expression E, we can define the *valuation* of E under I, denoted $[E]_I$, according to its syntactic structure. The valuation of E is either a Boolean value, an integer, or a function from integers to integers, according to whether E is a Boolean expression, an integer expression, or a function expression, respectively. A well-formed formula F is *true under interpretation* I if $[F]_I$ is TRUE. It is *valid* when it is true under all possible interpretations.

As explained earlier, the bounded model checker UCLID explores all execution sequences of the model presented to it of length up to the integer bound. On each step, the problem of checking the safety property is converted to that of checking the validity of a corresponding "safety formula" in the logic of Figure 12. UCLID performs the latter check by performing a validity-preserving translation of the safety formula to a Boolean satisfiability problem, which is checked using a SAT solver. Further details on UCLID's logic and decision procedure can be found elsewhere [10, 44].