Automatic Discovery of API-Level Exploits

Vinod Ganapathy[†], Sanjit A. Seshia[§], Somesh Jha[†], Thomas W. Reps[†], Randal E. Bryant[§]
[†]Computer Sciences Department
University of Wisconsin-Madison
Madison, WI-53706
[§]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA-15213
{vg,jha,reps}@cs.wisc.edu, {sanjit,bryant}@cs.cmu.edu

ABSTRACT

We argue that finding vulnerabilities in software components is different from finding exploits against them. Exploits that compromise security often use several low-level details of the component, such as layouts of stack frames. Existing software analysis tools, while effective at identifying vulnerabilities, fail to model low-level details, and are hence unsuitable for exploit-finding.

We study the issues involved in exploit-finding by considering application programming interface (API) level exploits. A software component is vulnerable to an API-level exploit if its security can be compromised by invoking a sequence of API operations allowed by the component. We present a framework to model low-level details of APIs, and develop an automatic technique based on bounded, infinite-state model checking to discover API-level exploits.

We present two instantiations of this framework. We show that format-string exploits can be modeled as API-level exploits, and demonstrate our technique by finding exploits against vulnerabilities in widely-used software. We also use the framework to model a cryptographic-key management API (the IBM CCA) and demonstrate a tool that identifies a previously known exploit.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification

General Terms: Algorithms, Security, Verification

Keywords: API-level exploit, bounded model checking

1. INTRODUCTION

A vulnerability in a software component is an error in its implementation that can possibly be used to alter the intended behavior of the component. An exploit is a sequence of operations that attacks the vulnerability, typically with malicious intent and devastating consequences. Recent years have witnessed a sharp increase in the number of security exploits. They are tricky to craft, because they often use several low-level details about the program's execution. For instance, a typical exploit against a buffer-overrun vulnerability uses details such as the layout of the stack, constraints on buffer sizes, and the architecture of the machine.

Supported by ONR contracts N00014-01-1-0796 and N00014-01-1-0708, and by ARO grant DAAD19-01-1-0485.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA
Copyright 2005 ACM 1-58113-963-2/05/0005 ...$5.00.

Given the growing concern over security, it is important to find exploits in a controlled environment before they are found and used by attackers. An analysis tool that finds a security exploit against a potential vulnerability in a component not only provides concrete evidence that the vulnerability exists, but also gives an analyst better insight into its consequences. For instance, static analyzers such as BOON [39] and Percent-S [33] would benefit from an analysis that finds exploits for vulnerabilities they identify. These tools produce false positives because of imprecision in their analysis, and the process of classifying warnings as real vulnerabilities or false positives is typically manual. A security exploit generated against a vulnerability identified by such tools offers several benefits. First, it provides evidence that the threat posed by the vulnerability is real. Second, the exploit can be used as a test case to stress the resilience of patched versions of the component. Finally, in cases where the analysis fails to produce a security exploit, the vulnerability can automatically be classified as a false positive, thus reducing the manual effort involved in classifying warnings.

Current tools do not adequately address the problem of finding security exploits in software. Tools based on model checking (e.g., [3, 10, 21]) have proved effective at finding control-flow-intensive vulnerabilities. However, these tools use finite-state abstractions, which abstract away details such as the layout of the program's stack and heap in the interest of keeping the analysis tractable. These very details are important to produce security exploits; as a result, counter-examples produced by these tools lack the detail to generate security exploits. Similarly, type-based analysis tools [18, 33] and constraint-based analysis tools [39] do not keep track of actual values of program variables. As a result, while these tools are effective at localizing vulnerabilities, they are not as effective at generating exploits against them.

In this paper, we study the issues involved in exploit-finding by considering API-level security exploits. A component is vulnerable to an API-level exploit if its security can be compromised by invoking an allowed sequence of operations from its API. For instance, the sequence seteuid(0) followed by execl(), allowed by UNIX, can be used to obtain `root` privileges [10].

We make the following contributions:

1. We present a formal framework to capture low-level details of API operations. The key idea is to abstract away as few details as possible, and produce a model that mimics the concrete system closely. The resulting model, which is typically infinite-state, is analyzed by model checking to determine if a state that violates a specified property is reachable. If so, the counter-example produced is translated into an exploit.
2. As an instantiation of the above framework, we consider two real-world APIs of significant complexity. We present a novel way to analyze `printf`-family format-string exploits as API-

level exploits, interpreting a format-string as a sequence of API operations. Reasoning about this API critically depends on modeling the runtime execution stack of the application precisely. We use the formal framework to model the above API, and demonstrate a tool to discover format-string exploits. We have used the tool to generate exploits against known vulnerabilities in real-world software packages. We also consider the use of our technique to analyze a subset of the IBM Common Cryptographic Architecture (CCA) API, which is a cryptographic key management API. In this case, it is crucial to model how an attacker can enhance his knowledge using operations from the API. Using a tool based on our technique, we discovered a previously known exploit.

3. Because our technique models data more precisely than existing tools [3, 10, 12, 21, 33], it is able to demonstrate the presence of a vulnerability by producing a security exploit that uses low-level details about the system. We demonstrate this by showing how our technique can find exploits against the vulnerabilities identified by Percent-S [33], a format-string vulnerability-finding tool. As discussed earlier, we demonstrate that finding exploits can benefit vulnerability-finding tools by automatically classifying vulnerabilities as real threats or as false positives.

The rest of this paper is organized as follows: We first present an overview of the technique in Section 2. We then present the formal framework in Section 3, and apply it to analyze `printf` (Section 4) and the IBM CCA API (Section 5). We discuss related work in Section 6, and conclude in Section 7.

2. OVERVIEW OF THE TECHNIQUE

In this section, we describe, using a toy protection system (Harrison *et al.* [20]), the framework used to specify APIs and the technique used to check such a specification. Section 4 on `printf`-family format-string attacks shows how the framework and checker can be used to generate security exploits that use low-level details, such as the organization of the program's runtime stack.

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.

Specifying the API. The first step in the analysis involves specifying the API in the formal framework, and specifying the safety property to be checked. The framework we use has four ingredients: (1) a set of variables that describe the state of the component that implements the API, (2) the initial state of the component, (3) the set of API operations and the semantics of these operations in terms of how they change the state of the component, and (4) a representation of the set of sequences of API operations to be checked. The fourth component helps to encode restrictions on the ordering of API commands. Such restrictions can be useful to exclude sequences of API commands that are inconsequential when analyzing the system, either because they can never arise in the execution of the system, or because the environment in which the system operates never generates such a sequence of API calls. When the set of sequences forms a regular language, this component can be expressed as a finite-state automaton.

As discussed earlier, the state of the protection system is described by the triple (*S, O, P*). The initial state of the protection system is given by the initial values of *S*, *O*, and *P*. Assume that these 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 three types of commands shown below with their semantics.

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

We assume that the protection system allows these operations to be applied in any order. Let us assume that we wish to check that the protection system obeys the security policy: "no subject can both read and write to an object that it does not own".

Checking the API. As explained earlier, to discover security exploits, it is important to work with the concrete system. Checking a finite-state abstraction often results in the loss of low-level details required to craft a security exploit. As a result, finding a security-exploit corresponds to checking the infinite-state system. For this purpose, we use *bounded model checking* [5]. An overview of the technique is shown in Figure 1.

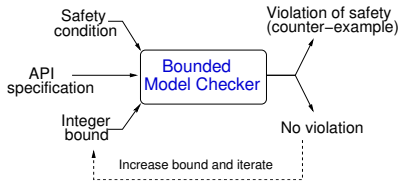

Figure 1: A schematic overview of the method.

The model checker accepts a description of the API specified in our framework, a safety property, and an integer bound. It systematically explores all allowed sequences of API operations shorter in length than the integer bound and determines whether any trace satisfies the safety condition. If the model checker finds a violation of the safety policy, it terminates with a trace of API operations that demonstrates the vulnerability. For instance, a bound of at least 3 discovers the following API-level vulnerability in the protection system: $\mathsf{Create}(\mathsf{A}, \texttt{file}) \rightarrow \mathsf{Confer}_{read}(\mathsf{A}, \mathsf{B}, \texttt{file}) \rightarrow \mathsf{Confer}_{write}(\mathsf{A}, \mathsf{B}, \texttt{file})$. This sequence of API operations adds (B, `file`, *read*), and (B, `file`, *write*) to *P*, but does not add (B, `file`, *own*). 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 exploits. Thus, in general, the iterative process could go on forever. Our procedure is *sound*, but *incomplete*. Thus, 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 component (the system to be analyzed) presents to client modules. Each command in the API changes the state of the component 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 individ-

ual API operations. We focus on such sequences of API operations, and how they affect the security of the underlying component.

Formally, a *component* $\mathcal{S}$ is defined by ($\mathcal{V}$, **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 $\vec{x} = (v_1, v_2, \ldots, v_n)$ is the *state* of the component $\mathcal{S}$. 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 component. Each state $\vec{x}$ such that **Init**($\vec{x}$) holds is a possible initial state of the component.
- 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 relations: $\mathsf{op}_i(\vec{a}_i) \subseteq \mathcal{D} \times \mathcal{D}$. 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 $\mathbf{Post}_i(\vec{a}_i)$: $\mathcal{D} \times \mathcal{D} \rightarrow$ BOOL, as: $\mathsf{op}_i(\vec{a}_i)(\vec{x}, \vec{y}) = \mathbf{Pre}_i(\vec{a}_i)(\vec{x}) \wedge \mathbf{Post}_i(\vec{a}_i)(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ denote, respectively, the state of $\mathcal{S}$ before and after the application of $\mathsf{op}_i(\vec{a}_i)$. If $\mathbf{Pre}_i(\vec{a}_i)(\vec{x})$ does not hold, then $\mathsf{op}_i(\vec{a}_i)$ aborts.
- $\mathcal{L} \subseteq \Sigma^*$ is a language of API operations. It plays two roles:

1. It encodes temporal restrictions on API operations that are inherent in the implementation of the component $\mathcal{S}$. This could, for example, be specified using a reference monitor.
2. It formalizes the notion of "usage patterns", i.e., API operation sequences that could be invoked by a client of $\mathcal{S}$. 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 exploit on the operating system. Rather than considering all possible sequences of system calls, it is sufficient to restrict our attention to call sequences that can be generated by the application [38].

Formally, $\mathcal{L}$ can be viewed as the intersection of two languages of API operations, one that plays the first role, and one that plays the second. The two-fold use of $\mathcal{L}$ is conceptually similar to the "optimistic" approach to interface design [13].

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 study in Section 4, we consider a special case of the framework presented above, in which $\mathcal{L}$ will be a regular language, and its recognizer will be a finite-state machine called the *API-automaton*.

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

In verifying that $\mathcal{S}$ never enters a state satisfying **Bad**, we restrict our attention only to sequences of API operations in $\mathcal{L}$. This avoids wasteful exploration of the state space during verification, and also reduces false alarms. Formally, we check for the following notion of *API-safety*:

DEFINITION 1 (API-SAFETY). For a predicate **Bad**, a component $\mathcal{S}$ is safe with respect to $\vec{x}$ if there is no satisfying assignment to the following formula for any finite value of k:

$$\exists\ \mathsf{op}_{i_1}, \mathsf{op}_{i_2}, \ldots, \mathsf{op}_{i_k}, \vec{a}_1, \vec{a}_2, \ldots, \vec{a}_k.$$
$$\mathbf{Init}(\vec{x}) \wedge (\mathsf{op}_{i_1} \cdot \mathsf{op}_{i_2} \cdot \ldots \cdot \mathsf{op}_{i_k} \in \mathcal{L}) \wedge$$
$$(\mathsf{op}_{i_1}(\vec{a}_1) \circ \mathsf{op}_{i_2}(\vec{a}_2) \circ \ldots \circ \mathsf{op}_{i_k}(\vec{a}_k))\ (\vec{x}, \vec{y}) \wedge \mathbf{Bad}(\vec{y})$$

'$\cdot$' denotes concatenation and '$\circ$' denotes relational composition, i.e., $(R_1 \circ R_2)(\vec{x}, \vec{z}) = \exists \vec{y}.\ R_1(\vec{x}, \vec{y}) \wedge R_2(\vec{y}, \vec{z})$.

An *API-level exploit* on the component $\mathcal{S}$ is defined as a sequence of API operations $\mathsf{op}_{i_1}, \mathsf{op}_{i_2}, \ldots, \mathsf{op}_{i_k}$, where $\mathsf{op}_{i_1} \cdot \mathsf{op}_{i_2} \cdot \ldots \cdot \mathsf{op}_{i_k} \in \mathcal{L}$, that violates API-safety of $\mathcal{S}$ for some predicate **Bad**.

Not surprisingly, for an arbitrary component $\mathcal{S}$ and predicate **Bad**, checking if $\mathcal{S}$ is safe with respect to state $\vec{x}$ is undecidable. The proof of undecidability follows easily from a similar theorem for protection systems [20].

Our approach to the API-safety problem is based on *bounded, infinite-state model checking*. To restrict attention to sequences of API operations in $\mathcal{L}$, we first construct the product of the language recognizer of $\mathcal{L}$ (e.g., the API-automaton) with the infinite-state system defined by ($\mathcal{V}$, **Init**, Σ). The safety property $\neg$**Bad** is then checked on the resulting infinite-state system $\mathcal{S}_{tot}$ using *bounded model checking*. The bounded model checker explores all API operation sequences of length up to an integer bound N in $\mathcal{S}_{tot}$, checking, for each state reached in that sequence, if **Bad** is satisfied. If so, it generates a *concrete error trace*; i.e., a sequence of states leading to the error state in which each variable v_i gets a value from the domain $\mathcal{D}_i$. An exploit is extracted from this concrete error trace.

While recent advances in SAT solving [29, 34] have made bounded model checking practical for analyzing finite-state systems, they have also led to the development of efficient SAT-based decision procedures for expressive, decidable first-order logics (e.g., [8, 14, 35]). This, in turn, has fueled progress in infinite-state bounded model checking, and is a key reason for our use of this technique.

3.1 Illustrative Example

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:

- $\mathcal{V}$ is $\{S, O, P\}$. 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** is ($S = O = \{\mathsf{A}, \mathsf{B}\}$) $\wedge$ ($P[\mathsf{A}, \mathsf{A}] = P[\mathsf{B}, \mathsf{B}] = \{own, read, write\}$) $\wedge$ ($P[\mathsf{A}, \mathsf{B}] = P[\mathsf{B}, \mathsf{A}] = \emptyset$).
- Σ is $\{\mathsf{Create}, \mathsf{Confer}_{read}, \mathsf{Confer}_{write}\}$. The predicate $\mathbf{Pre}(s, o)$ for $\mathsf{Create}(s, o)$ asserts that such an entry does not already exist in P, while $\mathbf{Post}(s, o)$ asserts that an entry (s, o) is created in P and $own \in P[s, o]$. The predicate $\mathbf{Pre}(s_1, s_2, o)$ for $\mathsf{Confer}_{read}(s_1, s_2, o)$ asserts that $own \in P[s_1, o]$, and $\mathbf{Post}(s_1, s_2, o)$ asserts that $read \in P[s_2, o]$. The predicates for Confer_{write} are similar.
- $\mathcal{L}$ is Σ^*. 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 $\mathcal{L}$ is a single-state finite-state machine with three transitions, one for each of the API operations in Σ. 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 discovers the exploit $\mathsf{Create} \rightarrow \mathsf{Confer}_{read} \rightarrow \mathsf{Confer}_{write}$.

4. FORMAT-STRING VULNERABILITIES

Format-string vulnerabilities [22, 30] 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*. While we restrict our discussion

to printf, the concepts discussed apply to other printf-family functions as well, e.g., syslog, sprintf. A format-string contains *conversion specifications*, which are instructions that specify the types that this call on printf expects for its 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 may be susceptible to a format-string vulnerability. 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.

Shankar *et al.* [33] have built a tool, Percent-S, to analyze source code and identify "tainted" format-strings that can be controlled by an attacker. Potentially vulnerable printf locations can also be identified in binary executables [22]. However, the aforementioned techniques do not produce format-strings that exploit the vulnerabilities they identify.

We present a novel way to analyze and understand printf-family format-string vulnerabilities. The format-string can be viewed 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 exploit is possible. If an exploit is possible, our tool produces a format-string that demonstrates the exploit. Our technique does not require the source code of the application and can analyze potentially vulnerable printf locations from binary executables. We have also used the tool in conjunction with Percent-S to generate format-strings that exploit the vulnerabilities identified (see Section 4.5). 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 placed on the stack from right to left. A call to foo(arg_1, arg_2) first places arg_2 on the stack, followed by arg_1. This is a popular C calling convention implemented by several compilers.
3. We analyze printf from the glibc-2.3 library.

4.1 Understanding printf

```
(1) int foo (char *usrinp) {
(2)    char fmt[LEN];
(3)    int a, b;
(4)    strncpy(fmt, usrinp, LEN - 1);
(5)    fmt[LEN - 1] = '\0';
(6)    printf(fmt);
(7) }
```

Figure 2: A procedure with a vulnerable call to printf.

This section reviews how printf works. Consider the code fragment shown in Figure 2. 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 3(A). In this case, printf is called with a pointer to fmt, which is a local character buffer in foo. This pointer is shown as the darkly shaded region in Figure 3(A).

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 3(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. printf 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 3(B). Note that ARGPTR advances only if the contents of fmt causes printf to enter argument-capture mode at least once.

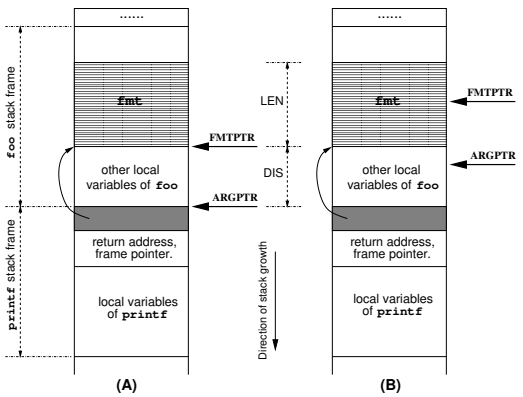

Figure 3: Runtime execution stack for the program in Figure 2.

To take a concrete example, suppose that fmt is "Hi%d" when printf is called in Figure 2. 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 2, 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 exploits discussed in this paper, 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`. Section 4.3 develops this point further.

4.2 Formally Specifying the `printf` API

The key observation is that each byte in the format-string is 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 specifies 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 exploit.

Each `printf` call is characterized by two parameters, namely the values DIS and LEN shown in Figure 3. The format-string vulnerabilities we consider occur 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 3 shows a simple scenario where the stack frame containing the format-string and the stack frame of `printf` 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 relevant details of the problem. Moreover, the values of DIS and LEN for each `printf` call can be obtained by disassembling the binary executable of the application that calls `printf`, and examining the call graph and the sizes of various functions' stack frames.

Formally, `printf` is described by $\mathcal{S} = (\mathcal{V}, \textbf{Init}, \Sigma, \mathcal{L})$, where:

- $\mathcal{V}$ denotes the set of local variables in the implementation of `printf` that capture the current state. We identified 24 local variables (or "flags") with integer and Boolean values[1] by examining the source code and manuals of `printf`. While our implementation considers all these flags, for purposes of explanation we restrict ourselves to just four flags: 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 initial values of the flags in $\mathcal{V}$. We assume that all addressing is relative to the initial location of ARGPTR, and hence **`Init`** is defined as (ARGPTR = 0) $\wedge$ (FMTPTR = DIS) $\wedge$ (DONE = 0) $\wedge$ (IS_LONGLONG = FALSE) for the four variables discussed here.
- Σ: As explained, each byte in the format-string is interpreted as an instruction to `printf`. 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 (corresponds to printing a **`"%"`** to `stdout`). Formally, [(MODE = printing) $\rightarrow$ (FMTPTR$'$ = FMTPTR + 1) $\wedge$ (MODE$'$ = argument-capture)] $\wedge$ [(MODE = argument-capture) $\rightarrow$ (FMTPTR$'$ = FMTPTR + 1) $\wedge$ (DONE$'$ = DONE + 1) $\wedge$ (MODE$'$ = printing)], where primed variables denote next-state values of the corresponding variables.
- We set $\mathcal{L}$ to be the language of all legal 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`.

There are several possibilities for **`Bad`**, each of which determines an attack that exploits format-string vulnerabilities. We present a few possibilities for **`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 implemented a tool to examine the above system and detect format-string exploits. The tool encodes `printf` as described above, and is parameterized by the values of DIS, LEN, and the predicate **`Bad`**. Our choice of a bounded model checker was 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 integer and Boolean variables, where the integer variables are modified using linear-arithmetic operations (addition and multiplication by a constant). To express formulas over these variables, we need quantifier-free Presburger arithmetic.

Based on these requirements, we chose to use the bounded model checking capabilities of the UCLID verifier. The details of how UCLID works are outside the scope of this paper, and may be found elsewhere [8, 32]. The description of `printf` ($\mathcal{S}$) 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 exploit. 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 Checking the `printf` API

In exploits 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.

Reading from an arbitrary location. One of the ways an attacker can print the contents of memory at address $a_4a_3a_2a_1$, 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

[1] 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 may lead to imprecision.

(A) **Bad** for Read Exploit	(B) **Bad** for Write Exploit
[FMTPTR $<$ DIS + (LEN $-$ 1) $-$ 1] $\wedge$ [ARGPTR $>$ DIS] $\wedge$ [ARGPTR $<$ DIS + (LEN $-$ 1) $-$ 4] $\wedge$ [*FMTPTR = '%'] $\wedge$ [*(FMTPTR + 1) = 's'] $\wedge$ [*ARGPTR = a_1] $\wedge$ [*(ARGPTR + 1) = a_2] $\wedge$ [*(ARGPTR + 2) = a_3] $\wedge$ [*(ARGPTR + 3) = a_4] $\wedge$ [MODE = printing]	[FMTPTR $<$ DIS + (LEN $-$ 1) $-$ 1] $\wedge$ [ARGPTR $>$ DIS] $\wedge$ [ARGPTR $<$ DIS + (LEN $-$ 1) $-$ 4] $\wedge$ [*FMTPTR = '%'] $\wedge$ [*(FMTPTR + 1) = 'n'] $\wedge$ [*ARGPTR = a_1] $\wedge$ [*(ARGPTR + 1) = a_2] $\wedge$ [*(ARGPTR + 2) = a_3] $\wedge$ [*(ARGPTR + 3) = a_4] $\wedge$ [DONE = WRITEVAL] $\wedge$ [MODE = printing]

Figure 4: The predicate `Bad` used for (A) Read exploit and (B) Write exploit.

the beginning of a sequence of 4 bytes, whose value as a pointer is $a_4a_3a_2a_1$.[2] Then, when `printf` reads the `"%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 4(A). Also, note that:

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; 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 show the need to track stack contents precisely.

Figure 5 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 exploit 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`. A few more observations on Figure 5:

1. In line (2), the tool is able to infer that an exploit is not possible. Intuitively, this is because the format-string is too small to contain a sequence of commands that carry out the exploit.
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 exploit as desired.

Writing to an arbitrary location. Another kind of format-string exploit 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 have been output by this call on `printf`. Figure 4(B) shows the case where an attacker writes the integer WRITEVAL to the address $a_4a_3a_2a_1$.

Figure 5 shows some format-strings obtained by the tool to write the integer 234 to memory address $a_4a_3a_2a_1$. 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 a width parameter, 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`, increments DONE by the value of WIDTH, 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 exploit.

The execution times shown in Figure 5 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, although not monotonically. The reason is that for larger values of LEN, it is necessary to run the bounded model checker UCLID for more steps, leading to a larger formula for it to check; the largest formulas were Boolean combinations of several thousand linear constraints over about a hundred integer variables. UCLID translates the problem into one of checking the validity of a Boolean formula, which we checked using a SAT solver called Siege [34]. Note also that the time taken for finding read exploits is much lower than that for finding write exploits. This is because finding a write exploit involves solving a more constrained problem than for the read exploit: 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 5). Furthermore, the length of this sequence can be at most LEN$-$1.

4.4 Optimizations

In our model of `printf`, *each byte* in the format-string is considered as an API operation. As an optimization we can add *aggregated API operations* to Σ, i.e., treat certain sequences of "primitive" API operations as a single operation. For example, we could create the aggregated API operation `"%Lg"`, which moves FMTPTR by 3 bytes, ARGPTR by 12 bytes, and reads a `long double` value. Similarly, we can use conservative width specifiers to form such an aggregate API operation;[3] e.g., `"%60Lg"` increments DONE by 60 in addition to changing the other flags as described above. Augmenting Σ in this way does not affect soundness because all the format-strings that UCLID could previously generate can still be generated. It is an optimization because longer strings can potentially be found with fewer iterations of bounded model checking.

4.5 Comparison with Existing Tools

To demonstrate the effectiveness of our tool, we compared it with Percent-S [33], a tool that analyzes source code using type-

[2] The only constraint on a_1, a_2, a_3, a_4 is that they must be non-zero, because a zero value is interpreted as `'\0'`, and terminates the format-string. For ease of explanation, we impose the additional restriction that $a_i \neq$ `"%"`, for $i \in \{1, 2, 3, 4\}$. If $a_i =$ `"%"`, the address can contain (parts of) a conversion specifier. However, our tool can also discover exploits where the address $a_4a_3a_2a_1$ contains `"%"`.

[3] The number of bytes printed is the maximum of the width specifier and that needed to precisely represent the output; so the width specifier must be conservatively large.

Sl.no.	DIS	LEN	Read exploit		Write exploit	
			Exploit string discovered	Time (sec.)	Exploit string discovered	Time (sec.)
(1)	0	7	"$a_1a_2a_3a_4$%s"	0.2	No exploit possible.	0.3
(2)	4	7	No exploit possible.	0.3	No exploit 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 5: Some exploits generated by our tool. For the write exploit, we chose to write the integer 234 to the memory location with a specific address $a_4a_3a_2a_1$.

Sl.no.	Software	DIS	LEN	Exploit	Exploit string discovered ("?" represents a non-zero non-% ASCII character)
(1)	php-3.0.16 [12]	24	1024	Write 0xbfff8cc3 to 0xbfff88c3 ($a_4a_3a_2a_1$) Assume that $b_4b_3b_2b_1 = a_4a_3a_2a_1 + 2$.	"$a_1a_2a_3a_4$%36000Lg%31Lg%n" + "?$b_1b_2b_3b_4$%13000Lg%111g%n"
(2)	qpopper-2.53 [31]	2120	1024	Read contents at 0xbfff88c3 ($a_4a_3a_2a_1$)	("%Lg")^{240} + ("?")^{52} + "%Ld%Ld%d%d%s$a_1a_2a_3a_4$"
(3)	wu-ftpd-2.6.0 [37]	9364	4096	Write 0xbfffbcab to 0xbfff88c3 ($a_4a_3a_2a_1$) Assume that $b_4b_3b_2b_1 = a_4a_3a_2a_1 + 2$.	"$a_1a_2a_3a_4$%99g$b_1b_2b_3b_4$" + ("%60Lg")^{778} + "%912g%600Lg%n%852X%n"

Figure 6: Exploits generated against known vulnerabilities in some real-world software packages.

qualifiers [18] to identify "tainted" (i.e., user-controlled) inputs that could potentially be used as format-strings. We report on two experiments here: the first showing how we can reduce the false alarm rate, and the second showing how we can confirm a true vulnerability by generating an exploit.

Consider the program in Figure 2. When compiled on our machine, the value of DIS is 28 bytes. Irrespective of the value of LEN, the size of the buffer `fmt`, Percent-S reports that the `printf` statement on line (6) is exploitable. Clearly, small values of LEN preclude the possibility of attack. As a result, Percent-S produces false alarms, because it does not account for the values of the parameters DIS and LEN.

On the other hand, using our model of `printf`, we were able to infer that a read-exploit (similar to the one reported earlier) is not possible unless LEN is at least 15 bytes, and a write-exploit (to write the integer 234) is not possible unless LEN is at least 20 bytes. In each of these cases, our analysis produces a format-string that demonstrates the exploit, while Percent-S does not.

We also used the tool to analyze known format-string vulnerabilities real-world software packages; Figure 6 has the details. `php-3.0.16` is a language-processor for the widely-used web-scripting language `php`, `qpopper-2.53` is a POP3 mail server, and `wu-ftpd-2.6.0` is a popular file-transfer daemon. We explain in detail the exploit against `wu-ftpd-2.6.0`; the others are similar. Percent-S correctly identified the location of the vulnerability in `wu-ftpd-2.6.0`, but did not produce a format-string demonstrating the exploit. The value of DIS and LEN for this example were 9364 and 4096, respectively, which we obtained by disassembling the binary executable. For these values of DIS and LEN, we checked whether the attacker could perform the following exploit: The attacker uses the buffer that stores the format-string to additionally store malicious code, and then overwrites the return address in the stack frame of `printf` using a write exploit (Section 4.3) so as to point to the beginning of the malicious code sequence instead. We assumed that the return address to be overwritten is at the stack location `0xbfff88c3`, and that the malicious code is located at the address `0xbfffbcab`, 13288 bytes above (and hence located within the buffer that stores the format-string). These address values are easily read off the stack using another exploit, as explained in Section 4.1. Because the value to be written is fairly large, we used a variant of the predicate **`Bad`** that allows for writing to a single address using multiple, slightly misaligned writes of smaller values. (Details on doing such misaligned writes can be found in [30, 37].)

Because the values of DIS and LEN are quite large, we had to use the optimizations described in Section 4.4. We were able to infer, in about 10 minutes, a format-string that is the concatenation of the following three strings: A prefix "$a_1a_2a_3a_4$%99g$b_1b_2b_3b_4$", a middle part ("%60Lg")^{778} consisting of 778 repetitions of an aggregated API operation, and a suffix "%912g%600Lg%n%852X%n", where $a_4a_3a_2a_1$ is 0xbfff88c3 and $b_4b_3b_2b_1 = a_4a_3a_2a_1 + 2$. It can be verified that this string writes the desired value to the desired location. One write is performed by each "%n": the first writes 0xbcab to $a_4a_3a_2a_1$ and the second writes 0xbfff to $b_4b_3b_2b_1$.

Existing format-string exploit generators attempt to construct format strings from fixed conversion specifiers. For instance, Thuemmel [37] constructs format-strings with the "%.8x" conversion specifier as the only building block. As a result, these techniques lack soundness: there may be exploit strings outside the space of strings explored by these tools. By doing an exhaustive search of the state space, our technique guarantees soundness within our model of `printf`. In addition, existing tools are incapable of finding variants of an exploit. As demonstrated in lines (3) and (4) of Figure 5, our technique can be used to discover variants of an exploit for the same values of DIS and LEN.

5. THE IBM CCA API

We present a second case study: the IBM CCA API. The CCA API [24] is a cryptographic-key management API used with secure hardware devices (coprocessors) such as the IBM 4758. The coprocessor provides cryptographic services, such as key-management, to the host computer. Each IBM 4758 is loaded with a distinct, secret *master key* (denoted by MK), which is safeguarded by the physical security of the device. The security of the host relies on the secrecy of MK.

5.1 Background on CCA

CCA is often used with the IBM 4758 for key-management. The host computer issues commands from the API, which are communicated by CCA to the coprocessor. A noteworthy feature of CCA is the ability to assign "types" (called *control vectors* [27]) to keys. The control vector of a key determines a subset of the operations from the API accessible to it. CCA defines control vectors for several classes of keys, such as those to compute message authentication codes, PIN numbers, and so on. Control vectors are used by CCA to implement role-based access control (RBAC) [17].

Because it is important to preserve the integrity of cryptographic keys, clear values of keys should never be stored on the host com-

puter. Similarly, to preserve the integrity of RBAC, a key should be tightly coupled with its control vector. The IBM 4758 achieves both these objectives by storing each key on the hard disk of the host computer as an *operational key-token*. For the discussion in this paper, we restrict ourselves to two components of the key-token, denoted as ($E_{MK \oplus CV_K}(K)$, CV_K). Here $\oplus$ denotes bit-wise exclusive-or, and $E_K(P)$ denotes the symmetric-key encryption (using an algorithm such as 3DES) of P using key K. Thus, the first component is the encrypted value of K, and the clear value of CV_K. When presented with this key token, the IBM 4758 can use CV_K from the second component, and use it to decrypt the first component to retrieve K. Of course, this clear value should not be revealed outside the IBM 4758. Observe that the value of K cannot be retrieved if the second component of the key-token is modified. Also note that this key-token will not function with another IBM 4758 because the master keys will be different.

It is often necessary for two hosts to share cryptographic-keys, for instance, to establish session-keys for communication. We discuss communication between two hosts A and B, each of which has an IBM 4758 (with keys MK_A and MK_B, respectively), and uses the CCA API for key management. One of the supported methods for communication involves establishing a secure communication channel between A and B, using a symmetric *key-encrypting key*, which is used to encrypt all CCA-managed keys transported over the channel. The key-encrypting key, whose clear value we denote as KEK, is itself a CCA key, and is associated with a control vector CV_{KEK}. It is stored at A and B as operational key-tokens ($E_{MK_A \oplus CV_{KEK}}(KEK)$, CV_{KEK}) and ($E_{MK_B \oplus CV_{KEK}}(KEK)$, CV_{KEK}), respectively. One of the techniques 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 KEK = $KP_1 \oplus KP_2$. These key parts are transported (in the clear) separately to B, where they are entered using Key_Part_Import, a CCA API-operation (see Figure 7). The result of this API-operation 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.

Consider a situation where A has a key (with control vector CV_K) stored as a key-token ($E_{MK_A \oplus CV_K}(K)$, CV_K), that it wants to share with B. Clearly, this key-token cannot directly be used by B because the clear value of K is encrypted with MK_A. To allow key-sharing between two IBM 4758s, CCA provides an API-operation Key_Export (shown in Figure 7) which makes the key-token "device-independent". This API-operation uses the operational key-tokens corresponding to KEK and K and to produce the token ($E_{KEK \oplus CV_K}(K)$, CV_K). This *export key-token* is device-independent. Intuitively, the key-token ($E_{MK_A \oplus CV_{KEK}}(KEK)$, CV_{KEK}) is used to retrieve the value KEK within the IBM 4758, which is then used to produce KEK$\oplus CV_K$, where CV_K is retrieved from key-token ($E_{MK_A \oplus CV_K}(K)$, CV_K). The IBM 4758 can also use ($E_{MK_A \oplus CV_K}(K)$, CV_K) to retrieve the value K. These values are 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-operation Key_Import (see specification in Figure 7) is used to convert this key-token into an operational key-token for B. The first input to this API-operation 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 CV_K from the second input to produce KEK$\oplus CV_K$. This value is used to retrieve K by decrypting $E_{KEK \oplus CV_K}(K)$. The clear value of K and the value of CV_K are then used to produce an operational key-token ($E_{MK_B \oplus CV_K}(K)$, CV_K), which can be used at B.

5.2 Formally Specifying the API

We 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) The host we analyze, A, can communicate with other hosts, such as B. (2) To do so, A and B establish a secure communication channel for key-exchange, protected by a key-encrypting key KEK. We assume that B initiates the communication, and the key-encrypting key is stored at B as ($E_{MK_B \oplus CV_{KEK}}(KEK)$, CV_{KEK}). (3) The API-operation Key_Part_Import is used to install key-encrypting keys.

Using the framework in Section 3, $\mathcal{S} = (\mathcal{V}, \texttt{Init}, \Sigma, \mathcal{L})$, where,

- $\mathcal{V}$ denotes a single set-valued variable $keytokens$, which denotes the set of all key-tokens known to A.
- **Init**: $keytokens = \emptyset$, the empty set.
- Σ = {Key_Part_Import, Key_Import, Key_Export}, i.e., the subset of the CCA API that we analyze.
- $\mathcal{L} = \Sigma^*$

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 assume that API-operations can be interleaved arbitrarily, 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(α, β): **Pre**(α, β) is TRUE while **Post**(α, β) is ($E_{MK_A \oplus CV_{KEK}}(\alpha \oplus \beta)$, CV_{KEK}) $\in keytokens$.
2. Key_Import(α, β): **Pre**(α, β) asserts that α and β have the structure of key-tokens. Let α^{enc} and α^{cv} denote the first half and second half, respectively, of key-token α, and similarly for β. **Post**(α, β) asserts that $E_{MK_A \oplus \beta^{cv}}(Key) \in keytokens$, where Key is such that $\beta^{enc} = E_{Val \oplus \beta^{cv}}(Key)$, and Val is such that $\alpha^{enc} = E_{MK_A \oplus \alpha^{cv}}(Val)$.
3. Key_Export: Analogous to Key_Import.

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

The safety property that we verify is the integrity of RBAC, i.e., the operational key-token obtained at A using Key_Import should be associated with the same control vector as the control vector associated with the export key-token sent by B. That is, if the value sent by B over the communication channel is ($E_{KEK \oplus CV_K}(K)$, CV_K), then the operational key-token at A must be ($E_{MK_A \oplus CV_K}(K)$, CV_K). **Bad** is defined as ($E_{MK_A \oplus CV_{new}}(K)$, CV_{new}) $\in keytokens$, where $CV_{new} \neq CV_K$, for some key K sent by B.

To study the security provided by the CCA API, we assume that an attacker has complete control over A. In particular, the attacker can observe and manipulate messages sent across the communication channel. In addition, he can manipulate any key-token stored on the host computer at A, and invoke CCA API operations on the IBM 4758 at A with arguments of his choice. These assumptions follow the standard Dolev-Yao attacker model [15]. A formal statement of the attacker's abilities is shown in Figure 8. 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 also knows $a \oplus b$.

5.3 Checking the API

We built a Prolog-based bounded model checker to analyze the above specification. We chose Prolog because the inference rules,

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

Figure 7: Some API operations from the IBM CCA. MK denotes the master key of the coprocessor CCA operates with, KEK denotes the clear value of the key-encrypting key, K denotes the clear value of a CCA key, CV_K denotes the control vector associated with K, and CV_{KEK} denotes the control vector for key-encrypting keys.

$$\oplus \text{ rules} : \frac{\Gamma \vdash a \quad \Gamma \vdash b}{\Gamma \vdash a \oplus b} \qquad \frac{\Gamma \vdash a \oplus b \quad \Gamma \vdash b}{\Gamma \vdash a}$$

$$\text{(En/De)cryption} : \frac{\Gamma \vdash k \quad \Gamma \vdash p}{\Gamma \vdash E_k(p)} \qquad \frac{\Gamma \vdash k \quad \Gamma \vdash E_k(p)}{\Gamma \vdash p}$$

$$\text{(Un)pairing} : \frac{\Gamma \vdash a \quad \Gamma \vdash b}{\Gamma \vdash (a,b)} \qquad \frac{\Gamma \vdash (a,b)}{\Gamma \vdash a \quad \Gamma \vdash b}$$

Figure 8: Knowledge enhancement rules. Associativity and commutativity rules for $\oplus$ are not shown.

such as those presented in Figure 8, and the API operations could easily be encoded as Prolog rules. We refer the reader to an accompanying technical report [19] for details on the model checker.

For the API specification discussed, the model checker produces the counter-example trace shown in Figure 9. This is the "chosen-difference" exploit on control vectors, first discovered by Bond [6].

(1) Key_Part_Import:
Input 1: KP_1
Input 2: $KP_2 \oplus CV_K \oplus CV_{new}$
Output: ($E_{MK_A \oplus CV_{KEK}}(KEK \oplus CV_K \oplus CV_{new})$, CV_{KEK})
(2) Key_Import:
Input 1: ($E_{MK_A \oplus CV_{KEK}}(KEK \oplus CV_K \oplus CV_{new})$, CV_{KEK})
Input 2: ($E_{KEK \oplus CV_K}(K)$, CV_{new})
Output: ($E_{MK_A \oplus CV_{new}}(K)$, CV_{new})

Figure 9: Counter-example trace showing exploit.

The exploit works as follows: Suppose that the attacker knows KP_2, where $KP_1 \oplus KP_2 = KEK$. This happens, for instance, when the attacker is the holder of KP_2. In statement (1) of Figure 9, the attacker installs a key of his choice as the key-encrypting key at A. Because the attacker can manipulate key part KP_2, he can produce $KP_2 \oplus CV_K \oplus CV_{new}$, 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 Key_Part_Import is executed with the modified key part as the second argument, the key-token ($E_{MK_A \oplus CV_{KEK}}(KEK \oplus CV_K \oplus CV_{new})$, CV_{KEK}) results, and A thinks that this is the key-token for the shared key-encrypting key. 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 ($E_{KEK \oplus CV_K}(K)$, CV_{new}) from ($E_{KEK \oplus CV_K}(K)$, CV_K), a value that he knows. Second, he invokes Key_Import with this modified key-token and the key-token of the shared key obtained in the first step of the attack. Key_Import produces $MK_A \oplus CV_{KEK}$ using the value of CV_{KEK} from Input 1, which is then used to retrieve $KEK \oplus CV_K \oplus CV_{new}$ from the first half of Input 1. Under normal operation this would have retrieved the value KEK instead. Key_Import then extracts CV_{new} from Input 2, and xor's this with $KEK \oplus CV_K \oplus CV_{new}$ to obtain $KEK \oplus CV_K$. This value is used to retrieve K from the portion $E_{KEK \oplus CV_K}(K)$ of Input 2. In the process, A has been fooled into thinking that the key is associated with the control vector CV_{new}. Hence, Key_Import terminates by producing an operational key-token ($E_{MK_A \oplus CV_{new}}(K)$, CV_{new}). This violates the integrity of RBAC, and completes the exploit; Bond [6] demonstrates how this can be used to learn sensitive values, such as PIN-encrypting keys.

It is worth noting that analyzing different APIs requires modeling different kinds of low-level details. For instance, in Section 4, we considered the layout of stack frames to discover format-string exploits. On the other hand, for the CCA API, we considered how an adversary could increase his knowledge using standard rules, such as those in Figure 8. We note that such rules are often employed by security-protocol verifiers (e.g., [28]), and the CCA API can potentially be analyzed by a security-protocol verifier as well.

6. RELATED WORK

Model Checking. Several software model checking tools (e.g., [3, 10, 11, 16, 21]) have been proposed in recent years. These tools check software for violations of user-defined assertions or of temporal-ordering rules on events. They use finite-state abstractions to model data values, and have been successful at verifying control-flow-intensive properties.

As discussed earlier, an API-level exploit is a concrete trace in the model that satisfies **Bad**. The key difference between an API-level exploit and concrete counter-examples produced by the above tools is that an exploit uses several low-level details. Unlike the aforementioned tools, our technique is capable of finding exploits because it permits modeling low-level details, such as the layout of the program's runtime stack.

The strategy we employ for finding exploits is based on bounded, infinite-state model checking. The use of the UCLID verifier in the `printf` case study was driven by the need to reason about quantifier-free Presburger arithmetic, and a bound of `LEN`-1 guarantees completeness. In general, the choice of an analysis tool would depend on the logic needed to reason about the system; e.g., if the underlying logic is first-order relational logic, the Alloy analyzer [23] could be used. Similarly, *unbounded*, infinite-state model checking techniques (e.g., [9]) can also be potentially used.

Test Generation. Formal specifications of software have been used to generate test cases, using bounded exhaustive testing [7, 36]. The specifications are typically in the form of pre- and post-conditions, and these tools exhaustively generate input data structures, upto a given size, that satisfy these conditions. Counterexamples produced by model checking tools [4] have also been used to generate test cases. Our analysis can also be viewed as a form of test generation. The API-level exploits generated can be used to test patched versions of the component that implements the API.

Ad-hoc Techniques. There is some prior work on security-exploit generators, including generators of format-string exploits [22, 30, 37]. However, as noted in Section 4.5, the techniques proposed are typically ad-hoc, and provide no soundness guarantees: they search only for specific attack patterns (e.g., format-strings using only a fixed conversion or width specifier), and hence might miss other kinds of attacks. In addition, these techniques are incapable of generating variants of an exploit. Thus, our paper presents a more general and formal framework that can generate exploits that previous exploit-generation tools cannot find.

Type- and Constraint-based Analysis. Static analyzers for special classes of vulnerabilities, such as buffer overruns (e.g., [39]) and format-string vulnerabilities (e.g., [33]) have also been proposed. As demonstrated in Section 4, our analysis complements such tools. An exploit generated against a vulnerability identified by these tools provides evidence that the vulnerability is real. On the other hand, if an exploit cannot be generated, the vulnerability can automatically be classified as a false alarm.

Interface Synthesis. Complementary to the analysis of finding

API-level exploits is the problem of synthesizing correct usage rules for APIs. Several techniques have been proposed to synthesize interfaces, including techniques that mine execution traces [2] and techniques based on model checking [1]. The output from these techniques, typically a finite-state machine over API operations, can be used as the API-automaton in our formal framework.

Superoptimizers. A code generator that produces code optimized with respect to certain criteria (e.g., number of instructions) is called a superoptimizer [26]. Recent work [25] has explored the use of SAT solvers and theorem-provers (using a technique similar to the one presented in Section 3) to produce superoptimized code. The space of possible code sequences is explored using a propositional Boolean formula, while the theorem prover is used to identify code sequences that satisfy the optimization criterion. Superoptimizers also model low-level instruction semantics; consequently, they often generate intricate, but compact, code sequences [26].

7. CONCLUSIONS

The main message of this paper is that it is necessary to model low-level details of a software component's implementation in order to find exploits against it. We demonstrated this by considering API-level exploits, and presented a framework to model and analyze APIs for exploits. We also showed the use of the framework by considering two real-world APIs of significant complexity. We briefly discuss some difficulties we encountered while modeling and analyzing APIs:

Modeling low-level details. As demonstrated by our case studies, different APIs can be exploited in different ways. Thus, the main problem is to identify the low-level details to model for each API. One possible solution is to model each API operation at the bit-level [11, 40], i.e., how each bit in the system is affected by applying the API operation. While this approach may solve the problem of identifying appropriate low-details for each API, it may not scale. We expect that, as with each of our case studies, the use of domain-specific expertise is the best solution to identify appropriate low-level details for each API.

Constructing the predicate `Bad`. Consider Figure 4: We used the fact that a `"%s"` can be used to read from a memory location. Consequently, the exploits found by our tool followed this blueprint. While this covers a large class of exploits, there may be other ways to read from memory, which our tool will miss. As before, domain-specific expertise is needed to construct an appropriate predicate **`Bad`** that covers a large class of exploits.

Automating model construction. The models of `printf` and the IBM CCA API were constructed manually by examining source code, and studying their manuals. This is clearly a tedious and error-prone process. Modern software model checkers [3, 21] automatically construct finite-state models using predicate abstraction. In our case, the main obstacles to automatic model construction are twofold: (1) low-level details to be modeled are domain-specific, and (2) the resulting model is often infinite-state (e.g., Section 4). In future work, we intend to investigate techniques, that, given the set of low-level details to be modeled, automatically extract models amenable for exploit-analysis.

8. REFERENCES

[1] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proc. 32nd POPL*. ACM, 2005.

[2] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc. 29th POPL*. ACM, 2002.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. 29th POPL*. ACM, 2002.

[4] D. Beyer, A. J. Chipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. 26th ICSE*. IEEE, 2004.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. 5th TACAS*, LNCS 1579. Springer, 1999.

[6] M. Bond. A chosen key difference attack on control vectors. *Manuscript*, November 2000. http://www.cl.cam.ac.uk/∼mkb23/research/CVDif.pdf.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. ISSTA*. ACM, 2002.

[8] 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 *Proc. 14th CAV*, LNCS 2404. Springer, 2002.

[9] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM TOPLAS*, 21(4):747–789, 1999.

[10] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Proc. 9th CCS*. ACM, 2002.

[11] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. 10th TACAS*, 2004.

[12] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from `printf` format-string vulnerabilities. In *Proc. 10th Security Symp.* USENIX, 2001.

[13] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. 8th ESEC and 9th FSE*. ACM, 2001.

[14] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proc. CADE*, LNCS 2392. Springer, 2002.

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

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th OSDI*. ACM/USENIX, 2000.

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

[18] J. S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *Proc. PLDI*. ACM, 1999.

[19] V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, and R. E. Bryant. Automatic discovery of API-level vulnerabilities. Technical Report 1512, CS Dept., Univ. of Wisconsin, 2004. http://www.cs.wisc.edu/wisa/papers/tr1512/tr1512.pdf.

[20] M. Harrison, W. Ruzzo, and J. Ullmann. Protection in operating systems. *Comm. ACM*, 19(8), 1976.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. 29th POPL*. ACM, 2002.

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

[23] D. Jackson. Automating first-order relational logic. In *Proc. FSE*. ACM, 2000.

[24] 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.

[25] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. In *Proc. PLDI*. ACM, 2002.

[26] H. Massalin. Superoptimizer—A look at the smallest program. In *Proc. 2nd ASPLOS*. ACM, 1987.

[27] 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.

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

[29] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 38th DAC*. ACM, 2001.

[30] T. Newsham. Format string attacks. www.securityfocus.com/guest/3342.

[31] SecurityFocus. Qualcomm qpopper vulnerability. www.securityfocus.com/advisories/2271.

[32] S. A. Seshia and R. E. Bryant. Deciding quantifier-free Presburger formulas using parameterized solution bounds. In *Proc. 19th LICS*. IEEE, 2004.

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

[34] Siege SAT solver. http://www.cs.sfu.ca/∼loryan/personal.

[35] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A cooperating validity checker. In *Proc. 14th CAV*, LNCS 2404. Springer, 2002.

[36] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *Proc. ISSTA*. ACM, 2004.

[37] A. Thuemmel. Analysis of format string bugs. *Manuscript*, 2001. http://downloads.securityfocus.com/library/format-bug-analysis.pdf.

[38] D. Wagner, and D. Dean Intrusion Detection via Static Analysis. In *Proc. Symp. on Security and Privacy*. IEEE, 2001.

[39] D. Wagner, J. S. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. NDSS*. ISOC, 2000.

[40] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *Proc. 32nd POPL*. ACM, 2005.