Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities

Richard Chang*, Guofei Jiang†, Franjo Ivančić†, Sriram Sankaranarayanan† and Vitaly Shmatikov*
*The University of Texas at Austin
†NEC Laboratories America

***Abstract*—As networked systems grow in complexity, they are increasingly vulnerable to denial-of-service (DoS) attacks involving resource exhaustion. A single malicious *input of coma* can trigger high-complexity behavior such as deep recursion in a carelessly implemented server, exhausting CPU time or stack space and making the server unavailable to legitimate clients. These DoS attacks exploit the semantics of the target application, are rarely associated with network traffic anomalies, and are thus extremely difficult to detect using conventional methods.**

We present SAFER, a static analysis tool for identifying potential DoS vulnerabilities and the root causes of resource-exhaustion attacks before the software is deployed. Our tool combines taint analysis with control dependency analysis to detect high-complexity control structures whose execution can be triggered by untrusted network inputs.

When evaluated on real-world networked applications, SAFER discovered previously unknown DoS vulnerabilities in the Expat XML parser and the SQLite library, as well as a new attack on a previously patched version of the wu-ftpd server. This demonstrates the importance of understanding and repairing the root causes of DoS vulnerabilities rather than simply blocking known malicious inputs.

I. INTRODUCTION

Complex networked systems are increasingly vulnerable to remote attacks that exploit unintended functionality and semantic implementation bugs. Denial of service is one of the most serious threats. For example, a single malicious packet containing an “input of death” [20], [5] can crash a server by exploiting a buffer overflow bug.

This paper focuses on sophisticated denial-of-service (DoS) attacks that deliberately cause resource exhaustion in networked applications. Typically, the attacker sends a single request or a small number of requests to trigger a computation that results in extraordinary consumption of an internal system resource such as CPU time or stack space. Because the malicious requests cause the target to freeze or even crash, we will call them *inputs of coma*.

Many popular systems have fallen victim to resource-exhaustion attacks, including Web servers [17], FTP [41], and DNS proxy servers [12], Samba [34], PHPMailer [31] and the Zend PHP engine [47], XML parsers [23] (the “billion laughs” XML attack also belongs to this class [16]), and, most recently, the Red Hat directory server [33].

Detecting software defects that can be exploited to cause denial of service is a challenging task. Unlike flooding and distributed DoS attacks, which inundate the target with a large number of requests, often originating from an orchestrated network of compromised machines (see the survey in [26]), inputs of coma are not associated with any network traffic anomalies and do not involve sending a large number of requests to the server. They can often be launched by a single attacker with a single malicious packet.

Unlike server-crashing inputs of death, inputs of coma do not rely on programming bugs that violate the semantics of the programming language in which the server is implemented. For example, buffer overflows violate memory safety or even control-flow integrity. By contrast, a maliciously crafted input that induces an exponential number (with respect to input size) of recursive calls to a regular-expression parsing routine in an FTP server (as is the case, for example, in the wu-ftpd vulnerability described below) does not violate any safety or integrity property. In fact, the target program does exactly what it was written to do; unfortunately, the implementers did not realize that their code can be abused to cause denial of service to legitimate clients.

Our goal is to develop a principled software analysis method that can detect DoS vulnerabilities *before* they are exploited by inputs of coma. The root causes of CPU, stack, and other resource-exhaustion vulnerabilities are often design flaws rather than programming errors. This observation motivates our approach. Our tool incorporates several resource-specific analyses which detect high-complexity control structures such as recursive calls and loops that potentially depend on tainted network inputs and can thus be exploited by a remote attacker to cause denial of service.

The *static* approach to discovering potential DoS vulnerabilities has several advantages. First, it helps identify and fix vulnerabilities during development, before the software is deployed, thus breaking the unwinnable loop of system administrators detecting and mitigating DoS attacks in real time as they are mounted against production servers. Second, conventional defenses against denial of service focus on network monitoring; adapting them to detect single-packet attacks that exploit the target application’s semantics in a non-trivial way is very difficult, if at all possible.

Our contributions. We present a novel static analysis approach to detecting semantic vulnerabilities in networked software which may be exploited to cause denial of service due to resource exhaustion. We focus in particular on CPU and stack exhaustion. Our approach has been implemented in a tool called SAFER: **S**tatic **A**nalysis **F**ramework for **E**xhaustion of **R**esources.

SAFER currently operates on C programs. We chose C as the target language because of its popularity for implementing networked applications and the fact that there are known resource-exhaustion DoS vulnerabilities in C applications. SAFER employs the CIL static analysis framework [28] and incorporates several heuristics for identifying loops and recursive calls whose execution is influenced by untrusted network inputs and for estimating their complexity.

We evaluated SAFER on several large applications, including FTP and web servers, an SQL library, and an XML parser. In addition to detecting known problems, SAFER uncovered **previously unknown denial-of-service vulnerabilities** in the Expat XML server and SQLite library, as well as a new exploit against wu-ftpd.

The latter exploit, which we will use as our running example, is particularly interesting. wu-ftpd was previously patched with a custom input sanitization to prevent precisely this kind of attack. Our analysis shows that the patch was insufficient, and a slightly more complex "input of coma" can be used to stage a DoS attack even on the patched version. This demonstrates the importance of identifying the *root causes* of DoS vulnerabilities–that is, the underlying software design flaws–rather than simply blocking specific attack inputs.

New wu-ftpd exploit. An example of a CPU exhaustion vulnerability appears in Listing 1, which shows the relevant portion of a pattern-matching function from the wu-ftpd implementation of an FTP server (the line numbers are for illustrative purposes only and do not correspond to the line numbers in the original code).

This pattern-matching code can be exploited by sending an input to the wu-ftpd server that causes `amatch()` to be called with arguments that induce a large number of recursive calls. These calls cause high CPU utilization and prevent other users from accessing the server. This attack, associated with the "`DIR *************...`" input, was discovered in 2005 [44]. It is worth noting that the initial vulnerability report placed it in the `wu_fnmatch()` function, rather than its true location, the body of `amatch()`. The pattern-matching code was subsequently patched with a custom input sanitization that collapses all consecutive wildcard symbols (`*`) prior to making the recursive call [42].

We applied SAFER to the latest version of wu-ftpd and, by analyzing its output, discovered a new attack input, "`DIR *{*{*{*{...}*}*}*}*`", that *works against the patched code*. (There can be no legitimate reason for a remote user to input this regular expression, which is equivalent to "`DIR *`".) This attack was not known prior to our analysis. Its denial-of-service effect is the same as in the original attack: the process reaches 100% CPU utilization for up to 10 minutes, making the server unavailable to legitimate users.

This example demonstrates several important features of semantic resource-exhaustion vulnerabilities, as well as our approach to detecting them. First, they are subtle. Focusing on specific attack inputs, such as the "`DIR *********...`" pattern above, may lead to a misinterpretation of the underlying vulnerability and leave the server exposed to denial-of-service attacks. This example also illustrates the danger of custom input sanitizations. A manual security audit of the source code, conducted without assistance from a tool like SAFER, might conclude that the code is safe because the input has been sanitized. Therefore, our analysis should be applied even to known and patched vulnerabilities, in case the patch proves insufficient.

Second, the attack does not involve a violation of memory safety, nor execution of any control paths not intended by the programmer. The pattern-matching function is implemented "correctly" in the sense that it correctly matches strings to regular expressions.

Third, other static analysis approaches (see Section II) are unlikely to detect this vulnerability. For example, in the case of static taint analyses, the set of source and sink calls is fixed independently of the program being analyzed. In the wu-ftpd exploit, however, a tainted network input does not flow into a fixed sink location; it is not obvious how to express the attack in terms of source and sink locations. This vulnerability is not easily expressed as a reachability property, either. It is not the case that simply reaching the recursive call leads to a DoS attack; the problem is that the number of recursive calls is a function of a tainted value. More specifically, the number of recursive calls is super-linear with respect to the input length.

```c
static int amatch(char *s, char *p)
{
    register int scc;
    int ok, lc;
    char *sgpathp;
    struct stat stb;
    int c, cc;
    globbed = 1;
    for (;;) {
        scc = *s++ & TRIM;
        switch (c = *p++) {
        case '{':
            return (execbrc(p - 1, s - 1));
        case '[':
            ok = 0;
            lc = 077777;
            while ((cc = *p++)) {
        ...
        case '*':
            if (!*p)
                return (1);
            if (*p == '/') {
                p++;
                goto slash;
            }
            s--;
            do {
/* exploitable recursive call */
                if (amatch(s, p))
                    return (1);
            } while (*s++);
            return (0);
        ...
        }
    }
}
```

Listing 1. wu-ftpd CPU Exhaustion Vulnerability

Structure of the paper. We discuss related work in Section II.

In Section III, we describe our approach and the implementation of SAFER. Section IV addresses the limitations of SAFER. In Section V, we discuss our experimental evaluation and describe new DoS vulnerabilities in real-world applications found by SAFER. Section VI concludes.

II. RELATED WORK

Defenses against DoS. Prior work on defenses against denial of service focused primarily on network-level detection of traffic anomalies and on filtering of malicious traffic [3], [19], [46], [39], [36]. While potentially effective against distributed DoS attacks, these approaches can only be activated once an attack has been mounted.

By contrast, we focus on a very different class of DoS vulnerabilities (relatively small, malicious inputs that cause resource exhaustion in server applications) and aim to discover their root causes by analyzing the source code of servers before they are deployed. While this paper is not about flooding attacks, we believe that the SAFER framework can be extended to detect *semantic* flooding attacks which try to exhaust system resources by exploiting the target program's semantics via a sequence of well-crafted inputs (see Section VI).

Qie *et al.* developed a toolkit for making software systems robust against DoS attacks that exhaust internal system resources [32]. This work is complementary to ours. The toolkit allows programmers to annotate program locations where resources are acquired and released, and to declare when resources should be reclaimed at runtime to recover from resource exhaustion. The programmer must analyze the code by hand in order to add the appropriate annotations. SAFER can be used in tandem with such a tool to guide programmers to potentially vulnerable code sections where resource-management annotations may be needed.

Inputs of coma are related to algorithmic-complexity DoS attacks that exhaust server resources by leveraging the disparity between average-case and worst-case behavior of certain server algorithms [10]. For example, a malicious sequence of inputs can be crafted to cause collisions in hash functions used to insert objects into data structures. Our approach is fundamentally different and complementary. While [10] relies on manual analysis of the hash-function implementation, SAFER *automatically* performs structural analysis of the *entire* source code, without assuming that the vulnerable behavior occurs in a particular routine. SAFER can be used, for example, to flag program locations where functions with potentially vulnerable worst-case behavior are invoked on tainted inputs; further analysis can then verify the presence of an algorithmic-complexity vulnerability.

Inputs of coma are much harder to detect at the network level than algorithmic-complexity attacks. The attacks against hash functions described in [10] require tens of thousands of network inputs. With inputs of coma, a single input is often sufficient for denial of service.

Recent work by Burnim *et al.* [4] proposes a testing-based approach for generating worst-case inputs for imperative programs and estimating worse-case computational complexity. Other techniques for estimating worst-case complexity include [8], [14]. These approaches are largely complementary to ours, which uses structural analysis to estimate complexity of loops and recursive calls. Unlike the techniques of [4], our analysis is static rather than dynamic; we do not require constraint generation and solving to explore individual program execution paths. Also, as demonstrated in Section V, our analyses scale to programs with more than 100,000 LOC, while the testing-based approach of [4] has been evaluated only on relatively small benchmarks such as sorting and tree-search algorithms. The static analysis framework underlying SAFER is general in the sense that it can incorporate new resource-specific analyses, including those that estimate computational complexity.

Formal methods can be used to identify computational asymmetries which may make security protocols vulnerable to denial of service [25]. This analysis is done at the level of protocol specifications, *i.e.*, at a much higher level of abstraction than the source-code-level analysis in this paper. Also, in the client-server applications of the kind of we consider (FTP, XML, HTTP), the server, by design, performs substantial computations in response to requests. The threshold beyond which an expensive computation becomes a denial-of-service attack is, fundamentally, a judgment call; there is a large "gray area" between obviously benign and obviously malicious inputs.

By contrast, our approach performs detailed static analysis of the source code of server applications in order to detect high-complexity control structures which may be exploited by malicious network inputs. This is a very complex class of software vulnerabilities which do not manifest themselves at the level of network-protocol specifications. Furthermore, our analysis highlights the vulnerable section(s) of the source code.

Zheng and Myers introduce a framework that uses static information-flow analysis to specify and enforce availability policies in programs [48]. They propose a way to specify availability policies as an extension to the decentralized label model, present a simple imperative language that allows explicit specification of security policies, and develop a security type system to reason about end-to-end availability policies. They also discuss several examples where violations of an availability policy correspond to denial-of-service attacks. This work targets a different domain than ours. We analyze real-world systems implemented in C and, unlike [48], do not require additional type annotations to be added to programs. Most importantly, our focus is on the root causes of resource-exhaustion attacks. To express these attacks as failures of some availability policy, it is necessary to model formally how resource exhaustion results in the failure of the system to deliver some expected output, which can be very difficult for reasonably complex systems.

Security applications of static analysis. Our SAFER tool utilizes many conventional static analysis tools and techniques, including the CIL front end [28], taint analysis, and dependency analysis. It is substantially different, however, from the existing program-analysis techniques, most of which are based on tainted data-dependency analysis or reachability analysis [24], [5].

Program-analysis tools have been used with great success to detect potential buffer overflows [38], [22], program crashes [5], and unsafe memory dereferences [29], [45], [13]. These tools tend to focus on violations of the intended program semantics, such as memory safety [11] or control-flow integrity [9], [1]. They cannot be used to detect semantic resource-exhaustion vulnerabilities, because the latter do not depend on such violations.

Security applications of taint analysis. *Taint analysis* techniques have been successfully used to detect many classes of security vulnerabilities. *Dynamic* taint analyses [7] monitor how untrusted inputs flow through the program during execution, and can be used at runtime to detect when a data value dependent on an untrusted input flows into a potentially dangerous function call or instruction without having been properly sanitized. This can be used, for example, to detect injection attacks [30]. Dynamic taint analysis can also be used at the instruction level [37], [21], [9] to prevent control transfers based on tainted data, such as those associated with buffer-overflow attacks.

Static taint analyses approximate the set of program variables that are data-dependent on untrusted inputs by statically analyzing the program's source code or compiled binary. Because of factors such as aliasing and polymorphic types, these analyses are often imprecise. Nevertheless, they have been successfully used to detect security vulnerabilities, such as cross-site scripting and SQL injection in web applications [24], [18], [40].

Taint analysis *cannot* be used directly to find DoS vulnerabilities because inputs of coma are not characterized by tainted values passed as arguments to certain functions. Instead, SAFER employs static taint analysis in a novel way by combining it with control dependency analysis to compute the set of program locations whose execution is influenced by tainted values (this is fundamentally different from "control hijacking" attacks, such as stack smashing and `return-to-libc`, because all control transfers are already present in the original code). In other words, we focus not on the use of values that are *data*-dependent on tainted inputs, but rather on the potential execution of basic blocks that are *control*-dependent on tainted inputs.

III. Static Detection of DoS Vulnerabilities

To detect vulnerabilities that allow a remote attacker to exhaust CPU or stack resources via inputs of coma, our SAFER framework uses a novel combination of two standard program analyses. First, taint analysis is used to compute the set of program values that are *data*-dependent on network inputs. Second, control dependency analysis is used to compute the set of program statements whose execution may affect whether or not a given statement is executed. SAFER combines the results of these analyses to compute the set of program statements whose execution is *control*-dependent on tainted values.

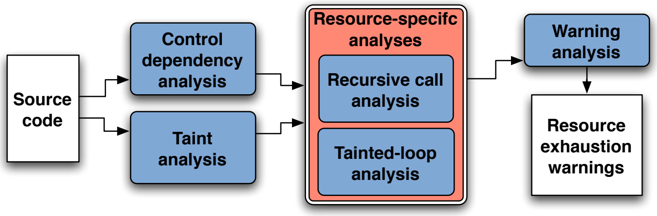

Fig. 1. SAFER Architecture

Figure 1 shows the architecture of SAFER. Our analysis focuses on loops and recursive calls because they (especially recursive calls inside loops) present the easiest targets for CPU- and stack-exhaustion attacks on C programs. First, SAFER computes the set of loops and recursive calls whose iterations and activation-record counts are potentially influenced by network inputs. Then structural analyses are used to estimate the complexity of recursion. The results are combined to generate warnings for all potentially exploitable recursive calls. Finally, the warnings are ranked by estimated complexity and severity before they are presented to the user.

We illustrate our framework by showing how each component analyzes the `amatch()` function from Listing 1. SAFER includes both intraprocedural components, which look at each function independently, and interprocedural components, which analyze the entire program, crossing function-call boundaries. Each component of SAFER is implemented as a CIL analysis built on top of several generic analyses provided by CIL [28]. In the following, many details of CIL's intermediate representation are simplified for expository clarity. For each component of our framework, we give an overview, the analysis algorithm used, and the result of analyzing the `amatch()` function.

A. Program representation

An imperative program can be represented by a set of procedures $P \cup Q$, wherein P represents the user-defined procedures, while Q represents calls to external procedures. Furthermore, there is a distinguished entry procedure main $\in P$ and a set of global variables G. Each procedure P consists of an intraprocedural control-flow graph (CFG) $(N, E, a, L, \mathsf{retVal}, \mathsf{entry}, \mathsf{exit})$, wherein N is a set of nodes corresponding to program locations, $E \subseteq N \times N$ is a set of edges, a is a function labeling each edge in the CFG with an *action*, L is a set of procedural formal and local variables, and $\mathsf{retVal} \in L$ is a special return value for the procedure. Finally, $\mathsf{entry}, \mathsf{exit} \in N$ denote the procedure entry and exits, respectively. Actions labeling edges include assignments to program variables, conditional guards, calls to other procedures in $P \cup Q$, and returns from the procedure.

To simplify the exposition, we assume that the programs are free from pointers and that procedures are called by value. This is done for expository clarity only; we do handle pointers in our implementation.

B. Control dependency analysis

SAFER employs a classic intraprocedural control dependency analysis which is based on computing post-dominators for each node in the control-flow graph of a function [27]. Informally, a CFG node s_1 is control-dependent on s_2 if whether s_1 is executed depends on the execution of conditional statements at s_2. In our `amatch()` example, the recursive call to `amatch()` at Line 29 is control-dependent on the `while` statement at Line 31 and the `case` statement at Line 19.

The result of control dependency analysis yields a map $\mathsf{CD} : P \times N \mapsto 2^N$, wherein $\mathsf{CD}(p, n)$ maps node n in a procedure p of the program to a subset of nodes in p, representing conditional statements on which n is control-dependent. Control dependencies can be computed in time linear in the size of the control-flow graph. This analysis is run on every function in the program and the resulting map can be queried for computing control dependencies.

C. Taint analysis

A program variable x is *tainted* at a node n in a procedure p under a particular context c if and only if there is some execution that reaches n under the context c such that the value of x is *data-dependent* on some *user input*.

The terms “data-dependent” and “user input” are at the heart of this definition and will be clarified below. Let I be the set of designated external procedures through which untrusted inputs arrive to the system (*e.g.*, I may contain the procedures that read data from network sockets). Let $q(a_1, \ldots, a_m) \in I$ be a procedure returning a value of type t, with possible side effects on some of its arguments $S_q \subseteq \{a_1, \ldots, a_m\}$ (for clarity, we will omit our analysis of pointers and side effects).

A *user input* to the program results from a call to an external procedure $q \in I$. Such a call yields two types of user inputs: (a) the return value of the call, and (b) the side effects, if any, that redefine global variables.

A variable x is said to be *data-dependent* on y at any program point if and only if there exists a *reaching definition* of the form $x := \mathbf{e}$ such that $\mathbf{e}$ is a program expression containing y, or some other variable z which is itself data-dependent on y.

Formally, we wish to compute a relation $T(c, p, n, x)$ denoting that a variable x is tainted at a node n in procedure p under a calling context c. This relation is computed using standard interprocedural program analysis, by applying inference rules until a fixed point is reached.

The rule external-input-taint (shown below) specifies that just after an assignment $x := q()$, the variable x may be regarded as tainted in the target node m under all contexts.

$$\frac{n \xrightarrow{x:=q()} m,\ q \in Q}{T(c, p, m, x)}\ (\mathsf{external-input-taint})$$

Some of the taint rules are shown in Fig. 2. The taint rule for assignment says that, as a result of an assignment $x := y$, if y is tainted at the source node, then x is tainted at the target. The call-taint rule says that, as a result of a call $r(y_1, \ldots, y_k)$ to an internal procedure $r \in P$, if y_i were tainted at the call site under some context c, then the corresponding formal argument arg_i is tainted at the entry node of the procedure r under the context $r :: c$. The return-taint rule governs the propagation of a tainted return value from a procedure back to its call site under an appropriate change of contexts. Our implementation uses additional rules for passing taints for variables which are not affected by a statement and also for handling variable aliasing due to pointers and call-by-reference. For brevity, we omit them here.

Implementing taint analysis. We defined taint analysis via a set of rules which specify the interprocedural, context-sensitive tainting relation $T(c, p, n, x)$. In practice, however, the number of contexts can be astronomical even for a medium-sized C program.

Therefore, we use *procedure summarization* to summarize the effect of a procedure call. Formally, a summary for a procedure p specifies the transformation of the relation $T(c, q, m, x)$ at any call site to p, $y := p(y_1, \ldots, y_m)$, for a variable x that may alias y, any of the arguments $y_1, \ldots, y_m$, or a global variable that is a side effect of the call to p. The summary for a procedure p expresses the possibility of a variable x being tainted at the exit $\mathsf{exit}(p)$ as a Boolean formula over atomic propositions that range over the taint-state of the program variables at the entry $\mathsf{entry}(p)$. The overall taint analysis consists of (a) bottom-up summarization and (b) top-down taint computation. The first phase computes the summary for each procedure assuming that all of its callees have already been summarized. This is enabled by performing a strongly-connected-component decomposition of the function call graph. Recursive procedures (including mutually recursive sets of procedures) are summarized by treating recursive calls *context-insensitively*. Each recursive call and each return from such a call is treated, effectively, as a *go-to* statement.

After each procedure is summarized, we revisit the call graph in the topological order, beginning at main and computing the actual taint relation T at the entry of each procedure and therefore at each node of each procedure. During this topological exploration of the call graph, we also record the set RF of visited procedures. The analysis that computes the taintedness relation T for each procedure is intraprocedural. Calls to other procedures are analyzed using summarization. We thus compute a relation $T(n, p, x)$, which is obtained by joining the earlier relation $T(c, n, p, x)$ under all possible calling contexts of p. We also define a Boolean relation $TS(n, p)$, which is true if and only if there exists some variable x such that there is a use of x on an outgoing edge from n and $T(n, p, x)$ is true.

The second phase can also be *query-driven*, *i.e.*, computed in response to queries asking whether a particular variable x is tainted under a particular context c. This is enabled by using the stored summaries and selectively analyzing the procedures called in the context c.

$$\frac{n \xrightarrow{x:=y} m,\ T(c,p,m,y)}{T(c,p,m,x)}\ (\mathsf{assign-taint})$$

$$\frac{n \xrightarrow{x:=r(y_1,\ldots,\mathbf{y_i},\ldots,y_k)} m,\ r \in P,\ T(c,p,n,y_i)}{T(r :: c, p, \mathsf{entry}(r), \mathsf{arg}_i)}\ (\mathsf{call-taint})$$

$$\frac{n \xrightarrow{x:=r(y)} m,\ r \in P,\ T(r :: c, r, \mathsf{exit}(r), \mathsf{retVal}(r))}{T(c,p,m,x)}\ (\mathsf{return-taint})$$

Fig. 2. Some rules for inferring and propagating taints across assignments, procedure calls and returns.

D. Warning generation

Our resource-specific analyses use taintedness and control-dependency information to estimate the complexity of recursive calls and to identify loops and recursive calls whose execution can be influenced by a remote attacker.

Tainted-loop analysis: Informally, we want to identify all loops for which the number of iterations depends on a tainted input. We statically approximate this set using the information computed by the previously described analyses. We say that any program loop in which there exists a path leaving the loop body whose final statement is control-dependent on a tainted statement inside the loop is a *tainted loop*. We compute the set TL of tainted loops via an intraprocedural analysis (implemented using CIL) which examines the control-flow graph (CFG) of each loop body in a given function. Note that the CIL front end detects syntactic loops in C programs and allows other CIL analyses to access CFG subgraphs corresponding to loop bodies.

For each loop body, we examine all edges that leave the loop body and check for tainted control dependencies. If any such dependencies exist within the loop body, we mark the loop as tainted and add it to TL. In the `amatch()` example, the `for` loop at Line 9 is marked as a tainted loop because there exists a path leaving the loop whose final statement is control-dependent on a `case` statement using `*p`, which has been marked as tainted. The algorithm for checking loop taintedness appears in Algorithm 1. SAFER executes it once for each set of CFG nodes corresponding to a top-level loop body in each function of the program.

```
Algorithm 1: $ComputeTL(p, L)$
Input: a set of CFG nodes $L$, and a procedure $p$
for $s \in L$ do
    if $isLoopHead(s)$ then
        $ComputeTL(LoopNodes(p, s))$;
    else
        for $(s, c) \in E$ do
            if $c \notin L$ then
                for $d \in CD(p, c)$ do
                    if $TS(d, p) \wedge d \in L$ then
                        $TL \leftarrow TL \cup \{L\}$;
                        break;
```

Recursive call analysis: We now describe the analyses used by SAFER to estimate the complexity of recursive calls and to identify those whose execution may be influenced by a remote attacker. SAFER first computes an *annotated call graph*, using an interprocedural analysis much like computing a standard call graph [27]. The key differences are that the SAFER call graphs are context-sensitive and annotated with additional information such as control dependencies, recursive calling contexts, taintedness, and source code information.

The algorithm for computing the annotated call graph appears in Algorithm 2. The graph consists of function nodes (with a unique node for every function in the program) and call nodes. The children of a function node are call nodes, which correspond to the calls in the function body. The call nodes contain information about the corresponding call sites (name of function, source code information, etc.). To compute the set of function nodes, SAFER uses the set of reachable functions RF computed during the taint analysis. SAFER also maintains a mapping from function names to function nodes in a map called *NameMap*. When computing the graph, SAFER also queries the taint analysis and control dependency analysis to check if a call is control-dependent on a tainted statement. We refer to such calls as *tainted calls*. Intuitively, their execution may be influenced by a remote attacker through tainted inputs.

After computing the annotated call graph, SAFER computes the set of tainted recursive calls and estimates their complexity. This analysis is performed via a bounded depth-first search of the annotated call graph.

Finding tainted recursive calls. We identify tainted recursive calls by performing a depth-first search on the annotated call graph while tracking the current calling context (Algorithm 3). Every time the search reaches a tainted call node, SAFER checks whether a call to the same function exists in the current context. If it does, the call node is recorded as a *tainted recursive call* and added to the corresponding set RC. We bound the search depth for efficiency (a large program can have an exponential number of calling contexts under which a particular call is recursive) and also because in our benchmarks, most recursive calls were found at very shallow search depths. Thus we limit the context depth to 3 in our experiments. An excerpt from the annotated call graph for wu-ftpd after identifying tainted recursive calls appears in Figure 3. As we explain in Section V, the recursive-calling-

Algorithm 2: *ComputeACG*

```
for $f \in RF$ do
    $t \leftarrow$ new FunctionNode;
    $t.name \leftarrow f$;
    $t.type \leftarrow$ function;
    $NameMap \leftarrow NameMap \cup (f, t)$;
    for $s \in CFG(f)$ do
        if $isCall(s)$ then
            $u \leftarrow$ new CallNode;
            $u.name \leftarrow calledFunc(f, s)$;
            $u.type \leftarrow$ call;
            $u.statement \leftarrow s$;
            $u.deps \leftarrow CD(f, s)$;
            $u.recContexts \leftarrow \emptyset$;
            for $d \in CD(f, s)$ do
                if $TS(d, f)$ then
                    $u.tainted \leftarrow$ true;
            $t.children \leftarrow t.children \cup \{u\}$
```

context data proved very useful during our manual analysis of the SAFER warnings and helped discover a new attack input.

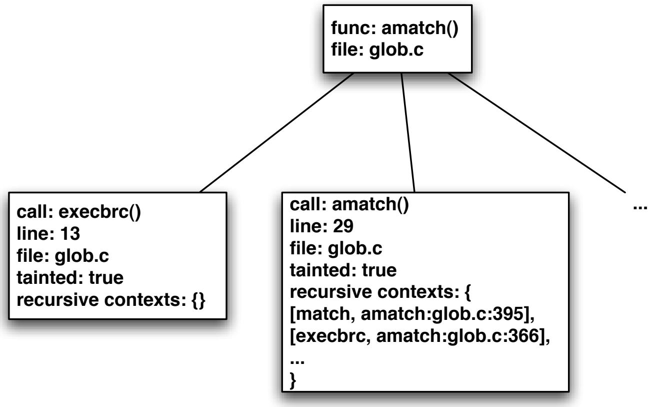

Fig. 3. Annotated Call Graph Node for `amatch()`

Estimating complexity of recursive calls. Attacks exploiting tainted recursive calls can lead to both CPU and stack exhaustion. Stack exhaustion occurs when a chain of recursive calls grows so deep that the program runs out of space for activation records, typically resulting in a crash. While these are serious vulnerabilities, their effect is limited to the vulnerable application. CPU exhaustion attacks can be even more dangerous because of their potential to affect other applications sharing the same server. If a single process begins to consume all available CPU resources, the result is denial of service to the clients of *all* applications.

Our running example, wu-ftpd, is a CPU exhaustion vulnerability. The depth of recursion is linear in the length of the attacker-controlled network input, thus stack space is not exhausted. Unfortunately, the tainted recursive call occurs inside a tainted loop. As a result, the total number of recursive calls generated by the vulnerable call site is exponential in the length of the attack input, leading to CPU exhaustion. (If one imagines the tree of recursive calls generated from this call site, its depth is linear, but the total number of nodes is exponential.)

Warnings caused by call structures of this type are of higher severity because of the potential for CPU exhaustion that could affect all processes on a server. When computing tainted recursive calls, SAFER also considers the information provided by the tainted-loop analysis. If a tainted recursive call occurs within a tainted loop, it generates a so-called *super-linear* warning, because of the potential for super-linear recursive behavior at this call site. Warnings of this type are collected in the SRC set (see Algorithm 4).

Algorithm 3: *ComputeRCWarnings*

```
for $f \in RF$ do
    $n \leftarrow NameMap[f]$;
    $C \leftarrow$ new Context;
    $FindTaintedRC(n, C)$;
```

Algorithm 4: $FindTaintedRC(n, C)$

```
Input: an annotated call graph node $n$, and a calling
        context $C$
if $Size(C) > ContextDepthLimit$ then
    return;
else
    if $n.type$ = function then
        for $p \in n.children$ do
            $FindTaintedRC(p, c)$;
    else                        /* n.type = call */
        if $n.tainted$ = true $\wedge C.contains(n.name)$ then
            $n.recContexts \leftarrow n.recContexts \cup C$;
            $RC \leftarrow RC \cup n$;
            for $L \in TL$ do
                if $n.statement \in L$ then
                    $SRC \leftarrow SRC \cup n$;
                    break;
        else
            $q \leftarrow NameMap[n.name]$;
            $FindTaintedRC(q, C.add(n.statement))$;
    return;
```

IV. LIMITATIONS AND TRADEOFFS

Analysis of potential resource-exhaustion vulnerabilities inevitably involves judgment calls. Unlike obvious coding bugs such as buffer overflows, software design flaws that make denial of service possible involve only legitimate memory accesses and execution of code paths which are already present in the original program.

The objective of SAFER is to flag high-complexity computations which can be triggered by untrusted network inputs. In many cases, they can be exploited to cause extremely high CPU utilization and/or deep recursion. Nevertheless, the distinction between a "legitimate" computation that just happens to freeze the server and a denial-of-service vulnerability is fundamentally a matter of opinion. The system developer must interpret the SAFER warnings. For example, imagine running our analysis on a server that decodes and streams video in response to client requests. By its very nature, it performs high-complexity computations whose execution is dependent on (tainted) client requests. Without developer's comments or annotations (*e.g.*, see [32]) indicating that these high-complexity code segments are benign even though they are control-dependent on tainted values, SAFER must report them as potential vulnerabilities. We employ user-tunable heuristics to minimize the number of warnings that are likely to be false positives.

SAFER deliberately sacrifices both soundness and completeness in order to scale to realistically-sized systems and detect real vulnerabilities. SAFER may miss vulnerabilities because it only supports a (realistic) subset of the C language (*e.g.*, we do not analyze function pointers). False positives may be caused by (i) imprecision of taint analysis, (ii) sanitization of user input, and (iii) safeguards on the usage of system resources.

Our taint analysis is not field-sensitive, which means that if any field in a complicated `struct` is marked as tainted, we assume all fields in that `struct` are tainted. This leads to some false positives where the analysis determines that an attacker may control the value of a field which is untainted in reality (see Listing 2).

```
APR_DECLARE(void) apr_pool_destroy(apr_pool_t *pool)
{
    ...
    while (pool->child)
/* pool->child is incorrectly marked
   as tainted because one of *pool's fields
   is data-dependent on a tainted value.
   recursive call below is a false positive. */
        apr_pool_destroy(pool->child);
    ...
}
```

Listing 2. Apache false positive (field insensitivity)

Known denial-of-service vulnerabilities, such as the original wu-ftpd vulnerability described in Section I, are often repaired by adding custom *sanitization* code, which checks and/or bounds the amount of computation due to tainted user inputs. For example, if the vulnerability is associated with reading a string from the network and then iterating over each character while performing an expensive computation at each iteration, then sanitization may attempt to limit the length of the string.

Traditional approaches for static reasoning about sanitization [24], [18] are syntactic and thus do not work for resource-exhaustion vulnerabilities. Balzarotti *et al.* [2] proposed a technique for reasoning about sanitization of tainted strings by modeling a set of strings as an automaton; a similar approach may work for sanitization with respect to CPU and stack resources. SAFER does not currently model sanitization of user input and may thus generate false positives for potential vulnerabilities which are not exploitable due to sanitization (see Listing 3).

```
int apr_fnmatch(char *p, char *string, int flags)
{
   ...
   for (stringstart = string;;) {
      switch (c = *p++) {
      ...
      case '*':
         c = *p;
/* input is sanitized by loop */
         while (c == '*') {
            c = *++p;
         }
         ...
         while ((test = *string) != EOS) {
/* recursive call below is a false positive */
            if (!apr_fnmatch(p, string, flags))
         ...
    }
}
```

Listing 3. Apache false positive (sanitization)

We emphasize that the mere presence of input sanitization does not mean that the code is safe and may give a false sense of security. As demonstrated by our exploit against *patched* wu-ftpd (discovered using SAFER), if sanitization simply blocks a specific attack input, the vulnerability remains exploitable by a different input. SAFER flags all functions in the code where sanitization is necessary; they must be audited to ensure that sanitization is present and effective. It is worth noting that the original report of the DoS vulnerability in wu-ftpd indicated the wrong function, while SAFER flags the correct one.

The third potential cause of false positives is the input-independent *safeguards* in the code that limit consumption of certain system resources. For example, SAFER may statically detect that a recursive call is "expensive" because it is dependent on a tainted value and inside a tainted loop, but the code may contain explicit (see Listing 5) or implicit (see Listing 4) safeguards to limit the depth of recursion.

The safeguards are handled in SAFER by adding user annotations for program-specific functions that check the height of a program's stack during execution. Several systems, including Apache, use special functions to help prevent some cases of stack exhaustion due to infinite recursion (often caused by misconfigurations). In the case of Apache, there is a risk of stack exhaustion if a request is recursively redirected a large number of times. This can occur if the server is misconfigured so that there is a cycle in the redirect rules. The basic idea is that while servicing a request via recursive calls, a function called `ap_is_recursion_limit_exceeded()` is called to check the current redirect recursion depth. This function does not actually check the current stack depth, but rather examines the request data structure which records this depth as a field. If the depth ever exceeds some statically set depth limit, then the request processing is aborted.

SAFER suppresses all warnings in which the recursive calling context contains a call to one of the user-defined stack-checking functions. Safeguards of this type are fundamentally different from user-input sanitization. They are designed specifically to detect when a program's stack is about to

```
int ap_rgetline_core(char **s, apr_size_t n,
                     apr_size_t *read, request_rec *r,
                     int fold, apr_bucket_brigade *bb)
{
    ...
    if (fold && bytes_handled && !saw_eos) {
        for (;;) {
            ...
/* False positive due to implicit limit
   on recursive call depth. Input is not sanitized,
   but the chain of recursive calls returns when a
   buffer is full. Therefore, the worst-case depth of
   calls is a function of this buffer size, not
   the tainted input. */
           rv = ap_rgetline_core(&tmp, next_size,
                                 &next_len, r, 0, bb);
           ...
}
```

Listing 4. Apache false positive (implicit safeguard)

be exhausted and to allow the program to gracefully handle the condition. The presence of a call to a stack-checking function on the execution path means—assuming it has been implemented correctly—that the developers are aware of the potential problem, eliminating the need for SAFER warnings.

The developer must be careful to place calls to stack-checking functions in *all* recursive calling contexts. If any tainted recursive call is found in a context without a safeguard, SAFER will generate a warning. It should also be noted that these stack-checking calls are often used to prevent only very specific types of stack exhaustion. For example, in Apache they only prevent stack exhaustion due to recursion when handling redirects. The reported warnings for Apache in Section V correspond to recursive calls that occur in contexts without stack-checking sanitization calls.

```
static int unixOpen(sqlite3_vfs *pVfs,
                    const char *zPath,
                    sqlite3_file *pFile, int flags,
                    int *pOutFlags) {
  ...
  fd = open(zName, oflags,
           isDelete?0600:DEFAULT);
  if(fd<0 && errno!=EISDIR && isReadWrite &&
     !isExclusive ) {
    flags &= ~(READWRITE|CREATE);
    flags |= READONLY;
 /* This call can only occur at call depth 0,
    all executions with more than 1 recursive call
    are infeasible because the call corresponds
    to simply attempting to open the file a second
    time with different permissions. */
    return unixOpen(pVfs,zPath,pFile,flags,pOutFlags);
  }
  ...
}
```

Listing 5. SQLite false positive (explicit safeguard)

Our practical experience with SAFER, described in Section V, demonstrates that SAFER can successfully discover complex DoS vulnerabilities in systems code, and that the number of false positives is relatively low for real-world programs (*e.g.*, 1 warning per 15,000 lines of Apache code) and thus amenable to manual analysis. The false positive ratio of SAFER is comparable to other static security-audit tools, such as those targeting buffer overflows [38], violations of Unix security rules [6] and injection vulnerabilities in web applications [18].

V. EXPERIMENTAL RESULTS

We evaluated SAFER on five large, real-world systems: Apache, wu-ftpd, Expat, SQLite, and Samba. The results of using SAFER to analyze these systems demonstrate that our approach scales to realistic programs. In 2 systems, SAFER rediscovered known DoS vulnerabilities; in 3 systems, SAFER discovered *new* DoS vulnerabilities.

A. wu-ftpd

wu-ftpd is a popular FTP server daemon available for Unix systems, including recent versions of Linux distributions such as Ubuntu [43]. We used SAFER to analyze the latest version (2.6.2) of wu-ftpd, which contains 20,382 LOC. SAFER reported 3 warnings, all of which correspond to recursive function calls where the depth of recursion is potentially super-linear in the size of a tainted input string. Two of the warnings were false positives, one due to the field-insensitivity of our taint analysis and the other due to input sanitization. The third warning was the `amatch()` call that we have been using as the running example throughout this paper.

wu-ftpd attack revisited. Recall the code in Listing 1. As described earlier, a DoS attack against wu-ftpd was discovered in February 2005 [44]. This attack results in CPU exhaustion due to an exponential number of recursive calls while trying to match the malicious "`DIR *********...`" pattern. The original mailing-list message attributed the vulnerability to a recursive call in the `wu_fnmatch()` function, but SAFER flags the recursive `amatch()` call, and our manual analysis confirms this call as the source of the vulnerability.

A patch designed to address this vulnerability was quickly incorporated into wu-ftpd packages used by Linux distributions [42]. The patch attempts to sanitize the input by collapsing contiguous blocks of wildcard symbols (*) into a single wildcard prior to performing pattern matching.

SAFER's context-sensitive recursive call analysis reports two contexts in which the recursive call is control-dependent on tainted inputs (see Figure 4 for a simplified warning). By examining both contexts, we discovered that sanitization can be nullified by a remote attacker. Our new attack input "`DIR *{*{*{*{...}*}*}*}*`" induces an exponential number of recursive calls to `amatch()`, just like the original attack.

```
Call funcName: amatch, file: glob.c, line: 448
Recursive calling contexts:
1) context start: matchdir
[match:glob.c:280, amatch:glob.c:395, ]
2) context start: expand
[execbrc:glob.c:240, amatch:glob.c:366, ]
```

Fig. 4. Simplified `amatch()` warning

In the first calling context, the sanitized pattern is passed to `amatch()` without modification. In the second context, however, after the sanitization code is executed, the pattern string is passed to `execbrc()`. With the new attack string, this function essentially "unsanitizes" it by removing matching braces, resulting in a string made up of wildcards (⋆) only. This string is then passed to `amatch()`, causing the same CPU exhaustion as the original attack. We have successfully confirmed our new "input of coma" against patched versions of wu-ftpd, including the latest version available for Ubuntu [43].

Observe that the input of coma in this case does not require the attacker to know anything about the file-system structure of the target. While some legitimate input patterns may trigger expensive computation in the pattern-matching routine (*e.g.*, those that match directories deep in the file system), our attack pattern is clearly malicious and can never occur in a benign FTP query.

This example highlights the subtlety of DoS attacks and how important it is to understand their root causes. The original vulnerability report cited an incorrect function as the cause and the sanitization patch only blocked a specific attack input. The fact that it could not prevent exploitation of the recursive call by a different "input of coma" demonstrates the need for a principled approach to detecting and preventing such attacks.

B. Expat

Expat is a stream-oriented XML parsing library that has been used by several open-source projects, including scripting-language implementations (Perl, Python, and PHP). We applied SAFER to the sample parsing application included with the Expat distribution of version 2.0.1 (12,251 LOC), along with the source code of the library itself. SAFER flagged 3 super-linear recursive call warnings. Our analysis of these warnings led to the discovery of a previously unknown stack-exhaustion attack against Expat involving the parsing of Document Type Definitions (DTD). A malicious remote user can induce arbitrarily deep recursion in an application that utilizes Expat by supplying a specially crafted DTD with a deeply nested element declaration. The application's call stack will then be exhausted, crashing the program.

```
static void
build_node(XML_Parser parser,
           int src_node,
           XML_Content *dest,
           XML_Content **contpos,
           XML_Char **strpos)
{
  ...
    for (i = 0,
         cn = dtd->scaffold[src_node].firstchild;
         i < dest->numchildren;
         i++, cn = dtd->scaffold[cn].nextsib) {
 /* recursive call that causes stack exhaustion
    when parsing DTD entity declaration with
    lots of nesting */
      build_node(parser, cn, &(dest->children[i]),
                 contpos, strpos);
    }
  ...
}
```

Listing 6. Expat Vulnerability

To check that a particular XML file is valid with respect to a given DTD, an XML parser must first parse the DTD itself. For any application that defines a handler for DTD elements, Expat parses the DTD via recursive calls to a function called `build_node()`, which generates an in-memory representation of a DTD. This recursive call is flagged as vulnerable by SAFER (see Listing 6).

Figure 5 shows a sample attack DTD. Attempting to parse this file with Expat leads to a recursive call chain whose depth is linear with respect to the number of nested elements. In practice, an 800-kilobyte DTD file of this form can crash the target application via stack exhaustion.

```
<!ELEMENT A (A,(A,(A,(A,(A,(A,(A,
(A,(A,(A,(A,(A,(A,(A,(A,(A,(A,(A,
(A,(A,(A,(A,(A,(A,(A,(A,(A,(A,(A,...
```

Fig. 5. Malicious DTD exploiting Expat vulnerability

C. SQLite

SQLite is a software library that implements a database engine. It has been used in many software systems as an application-file format, a web-application back end, and as a component of several operating systems for portable devices. For a recent version (3.6.3) consisting of 63,207 LOC, SAFER reported 6 super-linear recursive call warnings. Recall that super-linear warnings are more severe because of their potential to affect all processes on a system, rather than just the one in which the vulnerability occurs. The low number of super-linear warnings makes a detailed manual analysis feasible. One of the warnings turns out to be a potentially exploitable DoS vulnerability.

The fact that SQLite is a library and not a standalone application presents a slight challenge. SAFER uses taint analysis to model the ways in which a remote attacker may affect code execution, so normally the starting point would be some networked application that uses SQLite. We chose not to analyze any particular web application because it may only utilize a small part of the library. Instead, we analyzed the library along with a utility bundled with the SQLite distribution that allows to query a database from the command line. Therefore, we are implicitly assuming that all query strings are under attacker's control. The results produced by SAFER on this benchmark highlight complex recursive behavior in the library whose execution is triggered by database queries (note that in many web applications, the contents of database queries do depend on untrusted network inputs). This information may be used during security audit of the code to systematically identify all program locations where input safeguards or sanitization are needed, and to help the developers verify that all complex recursive behavior is bounded explicitly or implicitly.

The false-positive warnings for SQLite were due to sanitization. Many of them correspond to functions that recursively process in-memory query structures. The SQLite parser has several safeguards that prevent deeply nested queries from being processed by SQLite. To eliminate these false positives, we could have treated all values that pass through certain sanitization functions as safe (such functions would need to be annotated by the developer). This syntactic approach to modeling sanitization, however, can result in missing attacks, as in the wu-ftpd example.

We also note that these warnings are not false in the sense that they do correspond to recursive calls which are control-dependent on tainted input. Developers can thus focus their auditing and testing effort on ensuring that recursion is bounded in all of the flagged locations.

SQLite vulnerability. The code in Listing 7 contains a potentially exploitable recursive call inside the `sqlite3Select()` function. This recursive call is flagged as super-linear because it is inside the body of a tainted loop. A recursive call is generated on the call stack for each nested SELECT clause, and the `for` loop iterates over the tables in the FROM clause of the query. Consider the following query: `SELECT * FROM (SELECT * FROM (...) as t1) as t2 JOIN SELECT * FROM (`.... It attempts to maximize both the depth of the recursive call and the number of iterations of the loop.

```
SQLITE_PRIVATE int sqlite3Select(
  Parse *pParse,
  Select *p,
  SelectDest *pDest
){
  ...
  for(i=0; !p->pPrior && i<pTabList->nSrc; i++){
    ...
 /* recursive call that is vulnerable
    to CPU exhaustion */
      sqlite3Select(pParse, pSub, &dest);
      pItem->isPopulated = 1;
  ...
}
```

Listing 7. SQLite vulnerability

The safeguards in SQLite limit the nesting of `SELECT` clauses and prevent more than 64 tables from being joined in the `FROM` clause. These safeguards, however, still allow extremely large temporary results to be generated. Even for a small database consisting of a single table with 2 rows, a query of the above form could generate a result set with 2^{64} rows. In practice, iterating over this result set causes CPU exhaustion for several minutes. Note that the attacker does not need to know the database schema in order to mount this attack. Further analysis reveals that the recursive calls are not necessary for this attack, but the vulnerability was discovered due to warnings generated by Safer. The tainted loop allows an attacker to generate large result sets, with or without recursive-select subqueries. Because Safer computes the set of tainted loops for each program analyzed, we could have easily implemented an analysis that flags all tainted loops, such as the one in this example, but this would have produced many more false positives. One area of future work is incorporating range analysis [15] and ranking function synthesis [8] into our tainted-loop analysis in order to estimate bounds on loop iterations and use this information to filter out the false-positive warnings.

Our analysis of SQLite assumes that the attacker controls all query strings. This assumption may be unreasonable for well-written web applications, but if an application using SQLite as its back end were to fall victim to an SQL injection attack, this vulnerability can greatly increase the impact of the attack by allowing the attacker to exhaust CPU resources on the server. A conventional SQL injection attack only affects a single application and its database. An SQL injection combined with a resource-exhaustion attack on a co-hosted application may cause denial of service to all clients of all applications running on the shared server.

D. Apache HTTP Server

Apache is a widely deployed web server. It is the second largest application (109,650 LOC) that we analyzed using Safer, demonstrating the scalability of our approach. For a recent version (2.2.9) of Apache, Safer generated only 7 super-linear warnings, none of which correspond to DoS vulnerabilities. This false positive rate is substantially better than that of existing static analysis tools for finding potential security vulnerabilities [38], [6].

E. nmbd (Samba)

Samba is an open-source implementation of the SMB/CIFS protocol used to provide file and print services to networked clients. We analyzed the `nmbd` daemon which provides a NetBIOS name server as part of the 3.0.7 release of Samba. We chose this version because it contains a known CPU exhaustion vulnerability [34].

The `nmbd` daemon was the largest system we analyzed (131,662 LOC). Safer reported a total of 15 super-linear warnings. This again demonstrates the scalability of our approach. Most importantly, Safer did detect the known DoS vulnerability. The corresponding source code appears in Listing 8. This vulnerability is very similar in nature to the wu-ftpd vulnerability as it involves matching filenames with patterns containing contiguous wildcard (*) characters. Safer flagged two different pattern-matching routines that have essentially the same exponential recursive behavior. The original vulnerability report only mentions the attack input and the top-level pattern-matching function. A subsequent patch [35] modified the bodies of both functions to prevent the exponential recursion. While sanitization worked correctly in this case, Safer's analysis is able to identify the root causes of the vulnerability.

VI. Conclusions and Future Work

We have presented Safer, a static analysis tool for detecting potential resource-exhaustion vulnerabilities in networked software which can be exploited to cause denial of service. Safer uses novel resource-specific analyses which utilize taint- and control-dependency information to identify program branches whose execution can be influenced by a remote attacker via network inputs. Safer also identifies and

```
int ms_fnmatch_w(const smb_ucs2_t *pattern,
                 const smb_ucs2_t *string,
                 int protocol, BOOL c_s) {
  ...
  while ((c = *p++)) {
    switch (c) {
      ...
      case UCS2_CHAR('*'):
        for (; *n; n++) {
/* recursive call that is vulnerable to
   CPU exhaustion */
          if (!ms_fnmatch_w(p,n,protocol,c_s))
            return 0;
        }
        ...
}
```

Listing 8. Previously discovered Samba vulnerability

estimates the complexity of tainted recursive calls in order to detect CPU- and stack-exhaustion vulnerabilities. We have applied SAFER to several real-world systems, including FTP and HTTP servers and XML parsers. SAFER successfully discovered known DoS vulnerabilities, as well as three previously unknown ones.

Future work includes developing analyses for sanitization code, which can help filter out false positives, and extending our approach to other types of DoS vulnerabilities by augmenting SAFER with new resource-specific analyses for problems such as non-termination, memory leaks, and semantic flooding attacks that acquire and don't release resources (*e.g.*, TCP SYN floods which exhaust the server's thread pool). Other directions include automatic generation of attack inputs for the vulnerabilities discovered by SAFER and extending SAFER to languages such as PHP and Java.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS*, 2005.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in Web applications," in *S&P*, 2008.

[3] D. Bernstein, "SYN cookies," http://cr.yp.to/syncookies.html, 1996.

[4] J. Burnim, S. Juvekar, and K. Sen, "WISE: Automated test generation for worst-case complexity," in *ICSE*, 2009.

[5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically generating inputs of death," in *CCS*, 2006.

[6] H. Chen, D. Dean, and D. Wagner, "Model checking one million lines of C code," in *NDSS*, 2004.

[7] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *ISSTA*, 2007.

[8] M. Colón and H. Sipma, "Synthesis of linear ranking functions," in *TACAS*, 2001.

[9] J. Crandall and F. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *MICRO*, 2004.

[10] S. Crosby and D. Wallach, "Denial of service via algorithmic complexity attacks," in *USENIX Security*, 2003.

[11] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without runtime checks or garbage collection," *SIGPLAN Not.*, vol. 38, no. 7, pp. 69–80, 2003.

[12] "CVE-2005-2316," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-2316, 2005.

[13] D. Evans, J. Guttag, J. Horning, and Y. Tan, "LCLint: A tool for using specifications to check code," *SIGSOFT Softw. Eng. Notes*, vol. 19, no. 5, pp. 87–96, 1994.

[14] B. Gulavani and S. Gulwani, "A numerical abstract domain based on expression abstraction and max operator with application in timing analysis," in *CAV*, 2008.

[15] W. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Trans. Softw. Eng.*, vol. 3, no. 3, pp. 243–250, 1977.

[16] IBM, "Configure SAX parsers for secure processing," http://www.ibm.com/developerworks/xml/library/x-tipcfsx.html, 2005.

[17] "CVE-2003-0718," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0718, 2003.

[18] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting Web application vulnerabilities (short paper)," in *S&P*, 2006.

[19] S. Kandula, D. Katabi, M. Jacob, and A. Berger, "Botz-4-Sale: Surviving organized DDoS attacks that mimic flash crowds," in *NSDI*, 2005.

[20] M. Kenney, "Ping of Death," http://insecure.org/sploits/ping-o-death.html, 1997.

[21] J. Kong, C. Zou, and H. Zhou, "Improving software security via runtime instruction-level taint checking," in *ASID*, 2006.

[22] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *USENIX Security*, 2001.

[23] "CVE-2008-3281," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3281, 2008.

[24] B. Livshits and M. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *USENIX Security*, 2005.

[25] C. Meadows, "A formal framework and evaluation method for network denial of service," in *CSFW*, 1999.

[26] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 39–53, 2004.

[27] S. Muchnik, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[28] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *CC*, 2002.

[29] G. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *POPL*, 2002.

[30] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening Web applications using precise tainting," in *ISC*, 2005.

[31] "CVE-2005-1807," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1807, 2005.

[32] X. Qie, R. Pang, and L. Peterson, "Defensive programming: using an annotation toolkit to build DoS-resistant software," in *OSDI*, 2002.

[33] "CVE-2008-2930," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2930, 2008.

[34] "CVE-2004-0930," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0930, 2004.

[35] "samba-3.0.7-cve-2004-0930.patch," http://us5.samba.org/samba/ftp/patches/security/samba-3.0.7-CVE-2004-0930.patch, 2004.

[36] V. Sekar, N. Duffield, K. van der Merwe, O. Spatscheck, and H. Zhang, "LADS: Large-scale automated DDoS detection system," in *Proc. USENIX*, 2006.

[37] G. Suh, J. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ASPLOS*, 2004.

[38] D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *NDSS*, 2000.

[39] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker, "DDoS defense by offense," in *SIGCOMM*, 2006.

[40] G. Wassermann and Z. Su, "Sound and precise analysis of Web applications for injection vulnerabilities," in *PLDI*, 2007.

[41] "CVE-2005-0256," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0256, 2005.

[42] "Debian changelog wu-ftpd (2.6.2-20)," http://packages.debian.org/changelogs/pool/main/w/wu-ftpd/wu-ftpd_2.6.2-20/changelog.html#versionversion2.6.2-19, 2005.

[43] "Ubuntu – details of package wu-ftpd in gutsy," http://packages.ubuntu.com/gutsy/net/wu-ftpd, 2007.

[44] "WU-FTPD file globbing denial of service vulnerability," http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=207, February 2005.

[45] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors," in *ESEC/FSE*, 2003.

[46] X. Yang, D. Wetherall, and T. Anderson, "A DoS-limiting network architecture," in *SIGCOMM*, 2005.

[47] "CVE-2007-1285," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1285, 2007.

[48] L. Zheng and A. Myers, "End-to-end availability policies and noninterference," in *CSFW*, 2005.