6-14-2007

ABash: Finding Bugs in Bash Scripts

Karl Mazurak
University of Pennsylvania

Stephan A. Zdancewic
University of Pennsylvania, stevez@cis.upenn.edu

Karl Mazurak and Steve Zdancewic. **ABash: Finding Bugs in Bash Scripts**. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2007.
doi>10.1145/1255329.1255347

ABash: Finding Bugs in Bash Scripts

Abstract
This paper describes the design and implementation of ABash, a tool for statically analyzing programs written in the bash scripting language. Although it makes no formal guarantees against missed errors or spurious warnings (largely due to the highly dynamic nature of bash scripts), ABash is useful for detecting certain common program errors that may lead to security vulnerabilities. In experiments with 49 bash scripts taken from popular Internet repositories, ABash was able to identify 20 of them as containing bugs of varying severity while yielding only a reasonable number of spurious warnings on both these scripts and the generally bug-free initialization scripts of the Ubuntu Linux distribution. ABash works by performing *abstract interpretation* of a bash script via an abstract semantics that accounts for shell variable expansion. The analysis is also parameterized by a collection of signatures that describe external program interfaces (for Unix commands, etc.), yielding an easily configurable and extensible framework for finding bugs in bash scripts.

Disciplines
Computer Sciences

Comments
Karl Mazurak and Steve Zdancewic. **ABash: Finding Bugs in Bash Scripts**. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2007.

doi>10.1145/1255329.1255347

ABASH: Finding Bugs in Bash Scripts

Karl Mazurak
University of Pennsylvania
mazurak@cis.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

This paper describes the design and implementation of ABASH, a tool for statically analyzing programs written in the bash scripting language. Although it makes no formal guarantees against missed errors or spurious warnings (largely due to the highly dynamic nature of bash scripts), ABASH is useful for detecting certain common program errors that may lead to security vulnerabilities. In experiments with 49 bash scripts taken from popular Internet repositories, ABASH was able to identify 20 of them as containing bugs of varying severity while yielding only a reasonable number of spurious warnings on both these scripts and the generally bug-free initialization scripts of the Ubuntu Linux distribution. ABASH works by performing *abstract interpretation* of a bash script via an abstract semantics that accounts for shell variable expansion. The analysis is also parameterized by a collection of signatures that describe external program interfaces (for Unix commands, etc.), yielding an easily configurable and extensible framework for finding bugs in bash scripts.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Symbolic execution

General Terms Languages, Security, Experimentation

Keywords Bash, Scripting Languages, Abstract Interpretation

1. Introduction

Scripting languages are everywhere. For example, a typical Unix system employs dozens of shell scripts in the bootup process alone, and many more are used for everything from configuring firewall filters (e.g. via `iptables` under Linux) to processing CGI requests in web servers. Shell scripts are also an indispensable tool used by system administrators for routine maintenance and auditing.

Unfortunately, scripting languages are often designed for convenience over security. Their highly dynamic nature, their need for significant amounts of string processing, and their many interactions with external programs and system commands make scripts error-prone and difficult to debug. Most Unix systems wisely forbid scripts written in the traditional shell scripting languages (sh, csh, etc.) from being run setuid-root in order to avoid problems that may be caused by running buggy—or even perfectly well functioning—scripts. In practice, simple workarounds, such as writing a C wrapper program that calls the desired script, are sometimes used to sidestep this restriction; while the wrapper solution avoids certain bugs by running a script in a clean environment, it does not by any means eliminate all possibility of trouble. We are left in an unsatisfactory situation in which shell scripts are widely deployed despite widespread acknowledgment that they are prone to bugs that could lead to security problems.

This research aims to improve the situation with tools to help find bugs in shell scripts. To that end we present ABASH, a system for analyzing scripts written in bash (a common variant of the standard Unix sh). ABASH tracks abstract properties associated with program values, simulating an execution of the script; these properties can then be used to check both user-specified and automatically determined safety conditions. In particular, ABASH looks for situations where runtime string expansions might violate assumptions made by the script writer; such bugs could be used by an attacker to cause an external program to perform unexpected (and perhaps arbitrary) functions. This relatively simple analysis has allowed us to find many bugs in real-world bash scripts. Other static analyses can also fit into this framework; ABASH currently supports optional taint checking, allowing an administrator to specify that certain arguments to certain external programs must not be under direct user control, and other similar analyses could easily fit into this framework.

To summarize, this paper makes these two main contributions:

- We describe the design and implementation of ABASH, a tool for finding bugs in bash programs. Although ABASH does not guarantee the presence of errors or provably assert their absence, it still provides useful feedback that can point out common scripting errors. Implementation details are described in Section 3.
- We validate our implementation by running ABASH on a testbed of 49 bash scripts obtained from popular Internet repositories. In these experiments, we found 20 of these scripts to contain bugs of varying degrees of severity, suggesting that these bugs are indeed common in the wild. By contrast, a testbed consisting of initialization scripts from the Ubuntu Linux distribution exhibited no genuine bugs under our analysis, confirming the intuition that these are well-tested, carefully analyzed scripts. In both cases the number of spurious warnings is not unreasonable. These experiments are described in Section 4.

Of course, ABASH is not the first tool for analyzing scripting languages. Much of the related work has concentrated on languages for web scripting; Xie and Aiken's analysis of PhP [12] scripts is perhaps the closest to ours. That work and additional related work is discussed in Section 5.

Before diving into the technical details of our approach, we first present a brief overview of bash, focusing on the kinds of bugs detectable by ABASH.

PLAS'07 June 14, 2007, San Diego, California, USA.

2. Background: The bash scripting language

The bash (Bourne again shell) language is a member of the sh, csh, and ksh family of scripting languages, designed for automating routine tasks and "piping" together simple utility programs to create more complex behaviors. Bash is the default shell on most Linux systems as well as Macintosh OS X; it is available on Windows machines through the popular Cygwin POSIX emulation environment.

As with most shell scripting languages, Bash provides a "fork and exec" model of computation. After reading a line of text from the script (or from user input), bash breaks the text up into *words* following fairly complex quoting conventions that determine word boundaries. These words undergo a series of *expansions* that substitute values for variables and parameters, replace aliases with their definitions, interpret special characters (like `~`, which stands for the user's home directory), and split strings into words at white space. The shell then interprets the resulting words as *commands* followed by their arguments. Commands are either builtin bash operations, which are directly interpreted within the shell, or calls to external programs—perhaps other shell scripts—which are found by searching the `PATH` environment variable and run via the `exec` system call or its analog on the system in question. The inputs and outputs of the commands executed by the shell can be redirected to the terminal, files, or other processes to form pipelines of computations. Bash supports a standard suite of programming constructs: loops, conditionals, functions, arrays, variables, arithmetic operations, strings, and a limited form of pattern matching.

For the purposes of our work, the most significant step in the execution model described above is the expansion phase. The quoting mechanisms provided by bash interact with the expansion and substitution steps to determine whether any whitespace encountered should be included in the current word or seen as an indicator that the word is to be split into smaller words. If quotes are not used properly, a script may be vulnerable to attacks in which string values are interpreted in surprising ways, leading to unintended behavior; even non-malicious users may be surprised should they violate a script's unwritten assumptions.

As an example, consider a shell script that is intended to create a tar file of the directory `dir` and store it in the file named by variable `$1` with

```
tar czf $1 dir
```

If `$1` expands to the string `name.tar dir2`—that is, it expands to two words instead of one—the resulting command will be interpreted as:

```
tar czf name.tar dir2 dir
```

When executed, this command will include both `dir2` and `dir` in the resulting file `name.tar`; if this script were to be used, perhaps indirectly, by a web server in order to archive a pre-determined directory on a user's behalf, a malicious user could thus trick the script into producing an archive that also contained sensitive information from anywhere on the system that was accessible by the web server process.

This potentially serious flaw may be fixed quite easily be quoting he variable `$1`; the resulting script

```
tar czf "$1" dir
```

is now safe from this attack.

It is most certainly not the case that every variable should be quoted, however; often variables are meant to be split into multiple words. When moving a *list* of files specified by `$FILES` to a directory `$DIR`, `mv $FILES "$DIR"` works as one would hope, while `mv "$FILES" "$DIR"` is certainly wrong—in the latter case, the `mv` command would look for a single file with a name matching the space-separated concatenation of the intended filenames. As discussed in Section 3.4, the absence of such implicit typing information—in this case, whether the programmer intended a variable to represent a single file or a list of files—necessitates that we determine whether an expansion is intentional or accidental through heuristic means.

As a second, even more sinister example, consider the following invocation of the Perl interpreter[1]:

```
perl -e '<regexp>' $1
```

This example is a simplification of one found in one of our test cases, which attempts to extract timestamp information from files by matching against the elided regular expression. The problem is that the variable `$1` could potentially contain additional arguments that would be passed to `perl`—including a second `-e`, which could be used to execute arbitrary perl code embedded in `$1`. Simply quoting `$1` will not help in this case, as `-e` does not require a space before its code argument. The proper way to write this command in bash is as

```
perl -e '<regexp>' - "$1"
```

Here the `-` argument indicates that none of the arguments that follow should be interpreted as options; most programs for which this is relevant use either `-` or `--` for this purpose. However, shell scripts seen in the wild rarely use this safeguard, as we observe in Section 4.1.

In the main body of a bash script, the variable `$1` represents the first command line argument; thus, if a script containing either of the preceding examples were allowed to executed with privileges exceeding those of the person supplying the script's arguments—even if it was executed with a clean environment—it would present a security vulnerability that could lead to problems ranging from innocuous but unexpected behavior to the leakage or corruption of important files. In the case of the second example the situation might be even worse.

2.1 Challenges

Dynamic languages like bash are by their nature inclined to make static analysis difficult, but bash complicates with its simplicity in addition to complicating with its complexity. Intended more as "glue" than as a full-fledged programming language, bash scripts offload tasks to external programs as a matter of course, so, at some level, we must concern ourselves with the behavior of these programs.

Additionally, many syntactic elements of bash prove highly frustrating to the traditional lex-and-yacc-based approach to parsing source files. For example, most bash keywords are only treated specially in particular contexts, and one must at times track the depth of nested parentheses in order to understand how to correctly lex the current token. It is for this reason that, while ABASH handles quite a few constructs not found in other languages, we do not claim to support the entire bash language. We support a substantial subset of the language, however, and hope to expand to cover the corner cases in time.

3. Implementation

Our goal is to extract useful results from real-world bash scripts rather than to establish with certainty the soundness of scripts written in a limited subset of the language. Bash is a very dynamic language, and most, if not all, bash scripts exploit this; attempting to guarantee that a script is completely free of bugs—even if we

[1] Although Perl is a full-featured language in its own right, it is not uncommon to see it used this way from within scripts written in other languages.

s	::=	*string literals*	
v	::=	*variables*	
c	::=		*concrete words*
		sc	*strings*
		`${`v`}`c	*variables*
		`"`q`"`c	*quoted concrete words*
		‘*script*‘c	*command substitution*
		$\cdot$	*empty*
q	::=		*quotable concrete words*
		sq	*strings*
		`${`v`}`q	*variables*
		‘*script*‘q	*command substitution*
		$\cdot$	*empty*
script	::=		*bash scripts*
		stmt; *script*	*statement sequence*
		ϵ	*end of script*
stmt	::=		*statements*
		v`="`q`"`	*variable assignment*
		c *args*	*command*
args	::=		*arguments*
		c *args*	*argument sequence*
		ϵ	*end of arguments*

Figure 1. Syntax for concrete words and simplified bash scripts

limit our attention to the class of bugs described previously—would likely overwhelm the user with spurious warnings. We do not attempt to mitigate this problem with programmer annotations, in part because we want ABASH to be applicable to preexisting bash scripts, and in part because the effort required to add useful annotations may well exceed the effort required to identify and fix by hand the bugs that ABASH might discover. Instead, we identify expansion behavior that the programmer likely did not intend by using simple heuristics described in Section 3.4. It is our experience that these heuristics themselves are rarely responsible for spurious warnings; as noted in Section 4, ABASH's precision could be better improved by giving it more detailed knowledge of the intricacies of bash or by allowing it to reason about the output of various external commands.

Since our analysis focuses on a script's interaction with external programs, we would also like it to scale gracefully with the amount of effort administrators are willing to put into describing their systems. While a sysadmin may wish to manually annotate certain requirements of a few sensitive commands, it is unreasonable to expect this to be done for even a significant fraction of the programs on a typical Unix system. As we discuss in Section 3.4, ABASH evaluates the correctness of a call to an external program based on a *signature*; while signatures may be manually generated or edited, we provide both a simple utility that guesses signatures from `man` pages as well as a permissive default signature. ABASH generates reasonable output even when every external program is associated with the default signature, but adding explicit signatures allows us to identify more bugs.

3.1 Technical challenges and focus

Bash presents several challenges to formal analysis beyond those common to most scripting languages. In addition to those discussed in Section 2.1, bash deals largely with unquoted strings—which we refer to as *words* in keeping with the conventions of bash—

e	$\subseteq$	$\{0, 1, 2+\}$	*expansion levels*
t	::=	Tainted | Untainted	*taint levels*
l	::=	$\langle t, e \rangle$	*labels*
a	::=		*abstract words*
		sa	*strings*
		`${`v`}`a	*variables*
		`"`r`"`a	*quoted words*
		$?^l$	*unknown words*
		$\cdot$	*empty*
r	::=		*quotable abstract words*
		sr	*strings*
		`${`v`}`r	*variables*
		$?^l$	*unknown words*
		$\cdot$	*empty*
cmd	::=		*abstract commands*
		a	*command name*
		cmd a	*argument*

Figure 2. Syntax for labels and abstract words.

and these may expand in a wide variety of ways with sometimes surprising results.

Figure 1 gives a simplified syntax of bash words, which consist of concatenations of strings s, variable accesses `${`v`}`, quoted words, and embedded bash scripts. For the purposes of explaining our analysis, we restrict the set of scripts we consider to be sequences of simple statements, where a simple statement is either a variable assignment or a command invocation. (Loops and other more complex statement forms are discussed below.)

Our analysis focuses primarily on *parameter expansion*; that is, the replacement of variables by their values. Given that any variable not defined by the script itself may be inherited from the environment with an arbitrary value, this can be a major source of errors when the environment violates whatever assumptions the script writer might have been making. We alert the user to situations where potential parameter expansions are likely to violate the programmer's intent; we also optionally support taint checking, allowing certain arguments to an external program to be flagged if they might be unduly influenced by external sources.

Other varieties of expansion that seem unlikely to yield errors in our model receive less extensive treatment; while we do implement a simple evaluator for the expansion of arithmetic expressions, we assume that any command substitution (appearing in ‘back quotes‘) yields a completely arbitrary (and tainted) result, and we currently ignore brace and pathname pattern expansion. (The latter two would seem easy enough to handle given the correct separation of our analysis into multiple passes, but more accurately modeling command substitution requires reasoning about the output behavior of external programs, a thorny issue discussed further in Section 6.)

3.2 Implementation details

Our general approach has much in common with standard works in abstract interpretation [3], although rather than concerning ourselves with the final value of an abstract computation, we wish to extract from bash scripts a list of *abstract commands*. An abstract command in this case is simply one or more *abstract words*, the first denoting the command to be executed and the others its arguments. The syntax for abstract words is shown in Figure 2. An abstract word is a sequence of string literals, environment variables (which, unlike in concrete words, represent the initial value of that vari-

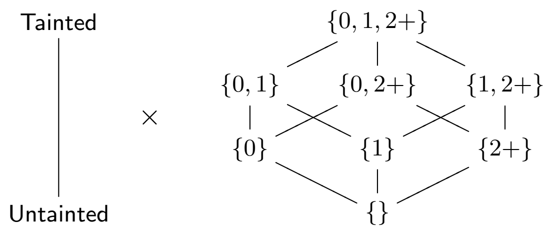

Figure 3. ABASH's label model: the product of an integrity lattice (left) and the expansion lattice (right).

able), and quoted abstract words ending in either the empty word $\cdot$ (which we will omit when it is unambiguous to do so) or an unknown word $?^l$, where l is a *label*. Note that a single abstract word may correspond to zero or more concrete words at execution time.

Labels are drawn from a simple product lattice, shown in Figure 3, with the first component being either $\mathsf{Tainted}$ or $\mathsf{Untainted}$ and the second a subset of $\{0, 1, 2+\}$.[2] The first component is standard for taint checking—tainted values are those which might be influenced in arbitrary ways by the user and thus cannot be trusted—while the second denotes the expansion properties of an abstract word. For example, an abstract word labeled with expansion component $\{1\}$ is guaranteed to correspond to exactly one concrete word at runtime, while an abstract word labeled with $\{0, 1, 2+\}$ may expand to a single concrete word, multiple concrete words, or no concrete words at all. We write $\top$ for the label $(\mathsf{Tainted}, \{0, 1, 2+\})$, describing tainted words with arbitrary expansion behavior; the corresponding $\bot = (\mathsf{Untainted}, \{\})$ describes no words due to its empty expansion component, but it is still useful in certain computations.

In addition to appearing within unknown words, a label describing a given word can be arrived at in a fairly straightforward manner; $\cdot$ has label $(\mathsf{Untainted}, \{0\})$, $?^l$ has label l, string literals have label $(\mathsf{Untainted}, \{1\})$ (a literal containing whitespace will not be parsed as such unless the whitespace is properly escaped.), and "a" has the same taintedness as a but an expansion component of $\{1\}$. Environment variables are by default labeled with $\top$, although in practice we allow the user to specify more precise labels for particular variables should this be appropriate. We write $\mathsf{label}(a)$ for the label of abstract word a, defined formally in Figure 4. It is these labels of abstract command arguments that are checked against command signatures as discussed in Section 3.4.

The extraction of abstract commands from our simplified bash scripts is for the most part straightforward; Figure 5 shows this process. The function $[\![script]\!]_{\mathcal{E}}$ yields that abstract interpretation of a given *script* in an abstract environment $\mathcal{E}$ mapping variables to quotable abstract words. The output of this function is a pair $(cmds, \mathcal{E}')$ consisting of the list of abstract commands *cmds* and a new environment $\mathcal{E}'$. This interpretation function simply threads the state of the environment through the sequence of statements that make up the script, updating the environment contents when a variable assignment is reached. We use the notation $cmds_1 +\!\!+ cmds_2$ to indicate the result of appending two lists of commands, and we use $[]$ for the empty list.

The abstraction of a concrete word in environment $\mathcal{E}$ is given by a function $[\![c]\!]_{\mathcal{E}}$, which yields a pair consisting of a list of abstract commands (arising because of embedded statements within

[2] In practice each component of a label is also annotated with a set of environment variables whose initial values went into the computation of the label. This provides users with valuable feedback about where the errors ABASH reports come from.

$$\mathsf{label}(?^l) = l \qquad \mathsf{label}(\$\{v\}) = \top$$

$$\mathsf{label}(s) = (\mathsf{Untainted}, \{1\}) \qquad \mathsf{label}(\cdot) = (\mathsf{Untainted}, \{0\})$$

$$\frac{\mathsf{label}(a) = (t, e)}{\mathsf{label}(\text{"}a\text{"}) = (t, \{1\})} \qquad \frac{\mathsf{label}(a_1) = l_1 \quad \mathsf{label}(a_2) = l_2}{\mathsf{label}(a_1 a_2) = l_1 \oplus l_2}$$

$$\begin{aligned} (t_1, e_1) \oplus (t_2, e_2) &= (t_1 \sqcup t_2, e_1 \oplus e_2) \\ e_1 \oplus e_2 &= \{\max(n_1, n_2) \mid n_1 \in e_1, n_2 \in e_2\} \end{aligned}$$

Figure 4. Labeling of abstract words

the word) and a list of abstract words that abstracts the concrete word itself. This definition is also straightforward. As ABASH does not currently model the output of external programs, the case for embedded statements simply extracts the underlying list of abstract commands and returns the single abstract word $?^\top$. Note that embedded statements that modify variables change only *local* copies of the variables, so the resulting environment is discarded.

The abstraction must also deal properly with quoting. The contents of variables that are mentioned outside of quotes must be split along whitespace boundaries; we write this function as $\mathsf{split}(a)$, which takes a single abstract word to a list of abstract words. It is used in the variable case of the $[\![c]\!]_{\mathcal{E}}$ function. Abstracting a quoted concrete word requires another function, $[\![q]\!]^q_{\mathcal{E}}$. It is defined similarly to $[\![c]\!]_{\mathcal{E}}$, except that it returns only a single abstract word (not a list) and the contents of variables read from the environment are not split at whitespace boundaries (since they are within quotes). The interesting case is for word concatenation: if the left word is statically known (i.e. its abstract value is not $?^l$), the result is a concatenated abstract word. If the left word is not statically known, however, the abstract word resulting from concatenation is itself unknown. This operation, written $a_1 :: a_2$ is defined by:

$$\begin{aligned} ?^l :: a &= ?^{l \oplus \mathsf{label}(a)} \\ a_1 :: a_2 &= a_1 a_2 \qquad \textit{when } a_1 \neq ?^l \end{aligned}$$

To concatenate two lists of abstract words, written $as_1 \frown as_2$, (needed in the definition of $[\![c_1 c_2]\!]_{\mathcal{E}}$), we merge the last element of the first list with the first element of the second list using the :: operation defined above:

$$\begin{aligned} as_1 \frown [] &= as_1 \\ [] \frown as_2 &= as_2 \\ a_0 \ldots a_n \frown a_{n+1} \ldots a_m &= a_0 \ldots (a_n :: a_{n+1}) \ldots a_m \end{aligned}$$

As we can see, it is not always straightforward to determine whether what is written as a single word is guaranteed to remain a single word at runtime. For simplicity we have chosen to require quotes around words assigned to variables, but in reality bash has no such requirement. Any such quotes, however, serve only to disambiguate parsing; one cannot, at the assignment of a string to a variable, protect the contents of a string from further splitting when the variable is used. This can come as quite a surprise to the novice bash programmer and can lead to exactly the sort of bugs that ABASH hopes to find.

To find these bugs, ABASH compares each abstract command in the interpretation of a script against command signatures that describe both the allowable expansion behaviors and desired taint restrictions for options and arguments to a given command; it also checks these arguments using heuristics assumed to be valid for *all* Unix commands. This process is described in Section 3.4.

$$[\![\cdot]\!]^q_{\mathcal{E}} = ([], \cdot) \qquad [\![s]\!]^q_{\mathcal{E}} = ([], s) \qquad [\![\texttt{\$\{}v\texttt{\}}]\!]^q_{\mathcal{E}} = ([], \mathcal{E}(v)) \qquad \frac{[\![script]\!]_{\mathcal{E}} = (cmds, \mathcal{E}')}{[\![\texttt{`}script\texttt{`}]\!]^q_{\mathcal{E}} = (cmds, ?^{\top})}$$

$$\frac{[\![q_1]\!]^q_{\mathcal{E}} = (cmds_1, r_1) \qquad [\![q_2]\!]^q_{\mathcal{E}} = (cmds_2, r_2)}{[\![q_1 q_2]\!]^q_{\mathcal{E}} = (cmds_1 +\!\!+ cmds_2, r_1 :: r_2)}$$

$$[\![\cdot]\!]_{\mathcal{E}} = ([], \cdot) \qquad [\![s]\!]_{\mathcal{E}} = ([], s) \qquad [\![\texttt{\$\{}v\texttt{\}}]\!]_{\mathcal{E}} = ([], \mathsf{split}(\mathcal{E}(v))) \qquad \frac{[\![q]\!]^q_{\mathcal{E}} = (cmds, r)}{[\![\texttt{"}q\texttt{"}]\!]_{\mathcal{E}} = (cmds, \texttt{"}r\texttt{"})} \qquad \frac{[\![script]\!]_{\mathcal{E}} = (cmds, \mathcal{E}')}{[\![\texttt{`}script\texttt{`}]\!]_{\mathcal{E}} = (cmds, ?^{\top})}$$

$$\frac{[\![c_1]\!]_{\mathcal{E}} = (cmds_1, as_1) \qquad [\![c_2]\!]_{\mathcal{E}} = (cmds_2, as_2)}{[\![c_1 c_2]\!]_{\mathcal{E}} = (cmds_1 +\!\!+ cmds_2, as_1 \frown as_2)}$$

$$\frac{[\![q]\!]^q_{\mathcal{E}} = (cmds, r)}{[\![v\texttt{="}q\texttt{"}]\!]_{\mathcal{E}} = (cmds, \mathcal{E}[v \to r])} \qquad \frac{[\![c_0]\!]_{\mathcal{E}} = (cmds_0, as_0) \quad \ldots \quad [\![c_n]\!]_{\mathcal{E}} = (cmds_n, as_n)}{[\![c_0\ c_1\ \ldots\ c_n]\!]_{\mathcal{E}} = (cmds_0 +\!\!+ \ldots +\!\!+ cmds_n +\!\!+ (as_0\ \ldots\ as_n), \mathcal{E})}$$

$$[\![\epsilon]\!]_{\mathcal{E}} = ([], \mathcal{E}) \qquad \frac{[\![stmt]\!]_{\mathcal{E}} = (cmds, \mathcal{E}') \qquad [\![script]\!]_{\mathcal{E}'} = (cmds', \mathcal{E}'')}{[\![stmt;\ script]\!]_{\mathcal{E}} = (cmds +\!\!+ cmds', \mathcal{E}'')}$$

Figure 5. The translation of concrete words to abstract words and the extraction of abstract commands

3.3 Technicalities

The full bash language introduces complications not present in the simple formalism described above. Therefore, before delving into the details of error detection, we first describe how the actual ABASH implementation deals these additional language constructs when creating the list of abstract commands for a given bash script.

Control and loops We have omitted control structures from our formalism, and, for the most part, they present no additional complications. The analysis of conventional conditionals (`if` statements, etc.) and less conventional features such as pipes between external commands proceeds structurally and in a generally straightforward manner. Bash does include several looping constructs, however, and like all static analysis tools ABASH must have some answer for blocks which may be executed an unknown number of times.

We choose to overapproximate the possible ramifications of a loop as follows: each loop bodies is analyzed twice; the first pass records any variables assigned to within the loop, and the second analyzes the body as normal under the assumption that all such variables evaluate to to $?^l$ for some placeholder label l. Commands are extracted normally in this second pass, which also constructs a directed graph of potential interdependencies among assigned-to variables. Labels are propagated along this graph, with $\top$ being used in the case of a cycle; the placeholder labels present in the extracted abstract commands are then replaced by labels taken from the graph.

Note that this is not the only possible approach to handling loops; rather than soundly overapproximating their effects, as we are, one could, for example, unroll loops a fixed (and small) number of times and assume that all relevant effects are accounted for. In practice, however, we have found that our approach does not lead to an unacceptable loss of precision—loops in bash scripts tend to be simple, and the ability to recognize variables with invariant labels despite their changing contents seems to be sufficient for our purposes. We have yet to see a real script in which greater precision in analyzing loops could lead to more accurate results.

ABASH does, in fact, perform loop unrolling when analyzing bash's `for` *var* `in` *list* loops, but only for the statically known prefix of *list*. This feature is disabled, however, when *list* consists entirely of values with the same label. In these cases loop unrolling is less likely to provide us with new information than it is to fill our output with repeated warnings[3].

Irrelevant commands In an effort to reduce spurious warnings from commands that cannot lead to security vulnerabilities, ABASH excludes from consideration several uninteresting commands along the lines of, for example, `echo`; the exact list may be specified by the user. We must take care, however, for while most uses of `echo` may be innocuous, others, like `echo -n 0 > /sys/some/kernel/module`, are most definitely not. Thus we ignore commands from our uninteresting set only when their output is not redirected and when they do not occur before any interesting commands in a pipeline. This allows us to ignore, for example, the `tee` command—used to send output to both standard out and and a file—and any uninteresting commands piped to it, which allows ABASH to produce much more readable output on scripts that employ this idiom.

Functions We defer interpretation of functions to their use sites, effectively inlining them everywhere. Although this does cause some repeated warnings, in our experience this redundancy is relatively minor in almost all cases—Section 4 describes the exceptions—and doing so allows us to easily and precisely analyze the effects of function calls. Functions in bash are bound dynamically and there are no restrictions on recursion, so it is quite possible to define functions that ABASH would be unable to analyze. Fortunately, these sorts of functions do not seem to appear in practice.

Builtins Bash has a multitude of builtins—commands that are executed by the shell directly and not spawned off as separate processes—some of which require attention in our analysis. Thankfully, most of these are relatively simple from our perspective. For example, `local` declares a local variable, while `read` assigns to variables from standard input (represented, of course, as $?^{\top}$). Others are more troublesome, however; the `set` builtin enables or disables a wide variety of options that alter bash's behavior, while `eval` treats an arbitrary string as bash syntax to be executed. Once again we are lucky in that these features are seldom used in real

[3] One script in our testbed iterates over each of 100 SCSI device nodes in the `/dev` filesystem on Solaris, running several commands on each node.

sig	::=	*name args opts*	*signatures*
args	::=		*argument schemas*
		arg args	*single argument*
		*arg**	*infinite arguments*
		ϵ	*no further arguments*
arg	::=	Tainted | Untainted	*arguments as partial labels*
opts	::=	*opt opts* | ϵ	*option lists*
opt	::=		*options*
		-name	*free-standing option (flag)*
		-name arg	*option taking a parameter*

Figure 6. Grammar for command signatures

world bash scripts—many of the troublesome builtins are meant for interactive use, or for inclusion in configuration files. (`eval` *is* used in practice, and we hope to eventually support it in ABASH; the main obstacle seems to be bash's varied and complex quoting conventions. At present ABASH simply warns unconditionally on any use of `eval`.)

Expansion Finally, bash's parameter expansion features a fair number of subtleties. Most important is the fact that `"$@"` (one way of referring to every parameter passed to a script or function) and `"${a[@]}"` (for array variables *a*) in fact expand to multiple quoted words whenever the array in question has more than one element. This breaks the invariant that a single quoted concrete word will necessarily be represented by a single quoted abstract word; ABASH tries to handle these cases properly, introducing a fair bit of complexity into the implementation.

3.4 Error detection

ABASH's error detection strategy uses a combination of heuristics and comparison of an abstract command and its list of abstract arguments against a signature that describes the expected expansion behavior and taint level for the arguments. This section describes the heuristics we use for error checking and for generating the command signatures.

Unfortunately, it can be difficult to tell whether a script writer expected a single source-level word to ever expand to multiple words at runtime. For example, splitting on the spaces present in `$FILES` may well be expected behavior, while similar splitting on `$FILE` is not; the latter should almost certainly be quoted. What then are we to make of `$FOO`? Although we are unable to address this problem in the general case, we do observe that, for example, `$FOO/$BAR` is almost certainly intended to expand to a single word. In fact, ABASH currently assumes that *any* static word formed directly from the concatenation of several word fragments is intended to expand to only a single dynamic word, and it warns when it cannot verify that such a word will not be split. In practice we have found that this is a very safe assumption to make; as seen in Section 4, spurious warnings due to intentional expansions of such words are extremely rare.

We also recognize that a concrete word appearing as the argument to an *option* should not expand such that only part of the word is taken as the option's argument; this allows us to rule out, for example, the misuse of `tar` seen in Section 1. In order to do this, however, we must know something about the external program in question. For example, `gcc -o $FOO` may go wrong if `$FOO` is split, but `ls -l $BAR` should elicit no such concern.

```
mkdir Tainted Tainted*
 -Z Untainted
 --context Untainted
 -m Untainted
 --mode Untainted
 -p
 --parents
 -v
 --verbose
 --help
 --version
```

Figure 7. Signature for `mkdir`

To this end, ABASH understands command *signatures*, which consist of the command name, a list of recognized options (and the parameters they may carry), and a schema for the remaining arguments to the command. The formal grammar for command signatures can be seen in Figure 6. Arguments may be annotated with the taintedness component of a label, but as each argument is meant to represent a single concrete word at runtime, the expansion component is superfluous. At present, an argument schema consists of a list of arguments optionally terminated by an indicator that an unbounded number of further arguments is permissible; one can imagine allowing more complex schemas, although it is unclear how helpful this would be in practice.

As an example, a signature for the GNU version of `mkdir` is given in Figure 7. Aside from the `--help` and `--version` options (which are recognized by nearly all GNU utilities), `mkdir` supports both long and short names for each of its options. The first two option pairs take arguments representing respectively a security context (applicable only under certain variants of Linux) and a Unix permission mask; it might be reasonable for a sysadmin to require these arguments be untainted, forbidding user input from directly determining the permissions at which a directory is created.[4] By contrast, disallowing users from naming the directories they wish to create seems too restrictive, so the main arguments to the command (of which there must be at least one) are labeled as tainted.

ABASH understands the GNU option convention, which includes both short options, e.g. `-o` (which may be grouped; `-abc` is equivalent to `-a -b -c` as long neither `-a` nor `-b` may take a parameter), and long options, e.g. `--option`. Option arguments may be optional or mandatory; in the case of `mkdir` all option arguments were mandatory, but when an option is permitted to appear with or without an associated argument it may simply be listed in both forms. We also handle variations on this convention; for example, some commands perform no grouping and thus make no distinction between short and long options, while others (like `ps` and `tar`) do not mandate the leading dash. Our approach cannot, however, be easily extended to programs with their own unique argument conventions, such as `find`; we return to this problem in Section 6.

As simple as our signature syntax may appear, writing signature files for many standard Unix utilities, which have evolved over the years to accept vast numbers of options, would still be a tedious task. It is for this reason that we also provide a simple Perl script[5] which is able to generate signature files from `man` pages. It is of course quite dependent on heuristics, and since it simply scans for options and attempts to determine whether they take pa-

[4] In a system which tracked *implicit* taint flow this might seem too restrictive, but as ABASH does no such tracking, there remain many ways in which user input may indirectly affect such an argument without triggering a warning. We argue that this is quite desirable, at least in this context.

[5] Perhaps illustrating the value of scripting languages.

	Internet repositories	`init.d`
Total scripts	**49**	**60**
Average number of lines per script	109.2	77.6
Average number of warnings per script†	7.6	1.9
Scripts with expansion bugs	**20**	**0**
Minor	10	0
Major	13	0
Scripts giving spurious warnings	**11**	**11**
Shallow	5	6
Deep	7	6
Intentional	1	0
Scripts vulnerable to option insertion	**39**	**31**

†Calculated over scripts with at least one warning.

Figure 8. Experiment results

rameters, it never requires that a parameter be untainted, and it always provides the most general label scheme (allowing any number of Tainted* words) for the remaining arguments. Finally, we provide a permissive default signature for use when no signature file is present, which accepts any option with or without a parameter.

Given an appropriate signature, ABASH identifies the options and option parameters to an abstract command; for example, if `-o` or `--option` may carry a parameter, then `FOO` is identified as that parameter in `-oFOO` or `--option=FOO`, but it is only identified as such in `-o FOO` or `--option FOO` if said option *must* be given an parameter. These option parameters and the remaining arguments to the command are checked against the labels in the signature, and a warning is raised whenever the label of the abstract word in question is not compatible with the label in the signature; that is, if the label of the abstract word is not less than $(t, \{1\})$, where t is the taintedness component taken from the signature.

In addition, the process of signature checking may reveal other potentially problematic situations. One of these, the presence of variables that may or may not be interpreted as options, is of special concern and discussed in detail in Section 4.1.

4. Evaluation

We have tested ABASH on a testbed of 109 bash scripts; 60 of these are `init` scripts from the Ubuntu Linux distribution[6], while the remaining 49 were obtained from the popular free software repositories SourceForge[7] and freshmeat[8]. The first set we believe to be relatively bug-free; initialization scripts are often fairly straightforward, and these in particular come primarily from well-tested and long-standing software packages. The second set seemed more likely to yield errors, and we believe it to be more indicative of "custom utility" shell scripts in the wild. It is our guess that wrapper scripts for complex applications fall somewhere between these extremes; they are most likely better tested than custom-made utility scripts, but they are often much more complex than simple initialization scripts.

We ran our tests using a set of signature files obtained by running our heuristic signature generator on 85 standard Unix utilities as well as on those commands that we manually determined were responsible for the main functions of the scripts in question for which `man` pages were available. It is possible that we could identify more potential option argument errors by writing custom signature files; in particular, such custom signatures might require untainted values at certain locations, while our automatically generated signatures do not make use of this aspect of ABASH. We hope, however that our results show first that minimal administrator effort is needed before our ABASH can be of use.

It is interesting to note that option argument errors were rather less frequent than expansion errors detected by our concatenation-based heuristic; only 3 of the 20 erroneous scripts suffered from potentially problematic option expansion. It is difficult to tell whether this is due to the relative infrequency of option arguments (as compared to other arguments) or simply because programmers are less likely to make this sort of mistake. Additionally, none of these errors were due to incorrect usage of standard Unix utilities; they occurred only in invocations of the commands we manually selected. This may be due to the simplicity of the options usually passed to common utilities, to the fact that script writers are less likely to misuse the commands with which they are most familiar, or simply to the kinds of scripts present in our data.

We classify as *shallow* any spurious warning that could be eliminated by minor improvements to ABASH's logic. Observing the harmless nature of a *deep* misclassification, by contrast, requires an understanding of either the filesystem or the output behavior of external programs. An *intentional* spurious warning represents a situation in which the usually erroneous possible behavior seems to be intended by the script writer. Similarly, we classify as *minor* any potential error that could nonetheless be avoided by assuring that environment variables are set to sane values before the script is run, or that depends on portions of the filesystem normally writable only by the super-user; *major* errors are those that depend directly on user input or parts of the filesystem under user control.

Our results can be seen in Figure 8. Because of our treatment of function calls as everywhere inlined, a single logical error can lead to multiple reported errors, and even without functions there is often great similarity between the errors in a given script; we thus give our results in terms of simply the number of offending scripts. However, in no case was the number of warnings too great for the classification of warnings to be done by hand. The average number of warnings per script was calculated over only those scripts which exhibited at least one warning; there were two outliers among the web scripts which yielded 42 and 74 warnings—in both cases due to repeated function calls—aside from which nearly every script produced fewer than 10. Such outliers could be rectified with a more refined treatment of functions, perhaps caching warnings produced by each function body and suppressing warnings sufficiently similar to ones already seen, but as it stands ABASH remains quite usable.

Figure 9 shows the expansion warnings given by analyzing a script from GeDI[9], a diskless image management tool; it also shows the abstract commands run by the script, which ABASH prints to aid in debugging. The variable `$IMAGE` is set to `$IMAGE/$1` at the top of the script, and some of the listed commands are in conditionals which fail harmlessly if `$1` expands to multiple words—although in bash this does not cause a script to terminate early without an explicit test and `exit` statement. Some, however, are not, and `rm -rf $IMAGE/var/tmp` is among those. One can imagine that this script (which is actually an example included with the GeDI distribution) might be run by a setuid wrapper which clears the environment and sets `$IMAGE` to a safe initial value. A system administrator who set up such a wrapper, intending that certain users be able to administer these disk images without root access to the system, would have unwittingly give these users the ability to delete any file accessible by root, simply by passing the script an argument containing whitespace.

We observe that, as expected, the `init` scripts appear to be free from errors detectable by our analysis; the 11 spurious warning

[6] `http://www.ubuntu.com/`

[7] `http://sourceforge.net/`

[8] `http://freshmeat.net/`

[9] `http://gedi-tools.sourceforge.net/`

```
warning: '${IMAGE}/etc/hosts' (as '${IMAGE}/${1}/etc/hosts') looks more expansive than intended [1 vs. 1+]
warning: '${IMAGE}/etc/' (as '${IMAGE}/${1}/etc/') looks more expansive than intended [1 vs. 1+]
warning: '${IMAGE}/var/tmp' (as '${IMAGE}/${1}/var/tmp') looks more expansive than intended [1 vs. 1+]
warning: '${IMAGE}/var/tmp' (as '${IMAGE}/${1}/var/tmp') looks more expansive than intended [1 vs. 1+]
warning: '${IMAGE}/root/' (as '${IMAGE}/${1}/root/') looks more expansive than intended [1 vs. 1+]

cmp -s /etc/hosts ${IMAGE}/${1}/etc/hosts
cp /etc/hosts ${IMAGE}/${1}/etc/
rm -rf ${IMAGE}/${1}/var/tmp
ln -sf ../tmp ${IMAGE}/${1}/var/tmp
cp /root/.X* ${IMAGE}/${1}/root/
```

Figure 9. Example output

cases could be reduced somewhat by minor improvements to our tool, but are not cumbersome to check by hand. Bugs are more prevalent in the scripts obtained from web repositories. In addition to 11 more cases of spurious warnings, we detect 20 scripts with actual errors that could be exploited to cause unintended behavior; 13 of these scripts remain exploitable even in a clean environment. Most of these errors are found by exploiting our heuristic for identifying words that are most likely not intended to expand. Signature checking finds argument option errors in only 3 scripts, but it is worth noting that those errors are not trivial—that is, they affect the behavior of commands that are important to the script in which they appear—and only one of them can be eliminated by enforcing a clean environment.

4.1 A trickier case

ABASH emits warnings other than the expansion notices discussed above; while most of these are primarily for diagnostic purposes at the moment, there is one that can be indicative of serious security violations. ABASH warns whenever an argument to a command may contain unknown options; this alerts us, for example, to the `perl -e` exploit discussed in Section 2. Unfortunately, it is tedious to make use of these warnings; they are raised by nearly every bash script, and often overshadow all other output. The last line in Figure 8 indicates just how prevalent they are; all but 10 of our sample scripts from web repositories yield this alert, generally quite often, although it is not the case that all—or even most—of these warnings represent actual exploitable bugs.

That these warnings are so common is not an artifact of ABASH. Although several of the `init` scripts correctly make use of the `-` and `--` arguments to denote that what follows should not be interpreted as containing options, only 2 of the rest of our testbed do so. Thus, in almost every case where a command takes a non-constant argument, that argument is reported by ABASH as potentially hiding one or more options; whenever such an argument is controlled by the user (which is very often the case), this represents a legitimate source of modified behavior.

Sometimes, of course, this is intentional; a variable `$OPTIONS`, for example, is probably meant to provide this functionality. Quite often it is not, however, and while not all of these unintended openings for additional options are quite as damaging as the example from Section 2, they still represent real potential bugs. But with a space of real bugs so large it becomes imperative to reduce the number of non-useful instances of this warning. We discuss some ideas on how to do this in Section 6.

4.2 Performance

ABASH is implemented in OCaml, and comes in at just over 4500 lines of code. One source of significant complexity in the code is the difficulty of parsing bash: parenthesis matching must be taken into account during the lexing phase, and the set of recognized keywords is context sensitive—resulting in a convoluted and incomplete parser.

Although ABASH is not optimized for speed, it performs quite well, checking our entire testbed of 109 scripts (from parsing to signature checking) in under 5 seconds on a 2 GHz Pentium M laptop.

5. Related work

The research most closely related to the ABASH system is Xie and Aiken's algorithm for statically detecting security vulnerabilities in PHP scripts [12]. Their approach also uses abstract interpretation [3] to approximate the behavior of PHP scripts, focusing on a taint analysis. In contrast to our approach, in which we treat functions as though they are inlined at their call sites, Xie and Aiken's PHP analysis computes summaries of the functions and uses call-graph information to perform interprocedural analysis. In theory, the summary approach should be more scalable (and it also eliminates the possibility of the analysis going into an infinite loop), but we have found that our simple inlining approach works well in practice. The abstract values used in ABASH use a simple representation of the expansion levels to detect likely bugs in bash scripts, but, unlike Xie and Aiken's approach, it does not attempt to identify where "sanitization" of tainted inputs occurs (typically by matching against a regular expression). It is likely that ABASH would produce fewer spurious warnings when doing its taint-checking analysis by adopting this strategy. Ultimately, the motivations and goals for static analysis of PHP and bash are quite similar, and, although there are significant differences between these scripting languages, we feel that techniques developed in one context are likely to be applicable in the other.

There have been many other applications of static analyses (in the form of type checking or model checking) to find security flaws in software. Some prominent examples of C analysis tools include CQual [7] and SLAM [1], and there has also been work on analyzing Java to find security vulnerabilities (see, for example the work by Livshits and Lam [8]), for static verification (ESC/Java [6]), and to enforce information flow policies (see Jif, for example [9]). Su and Wasserman provide a semantic, language-independent assessment of command injection attacks based parse trees [11]. Their work has been applied in the context of SQL injection attacks against web applications. Chen and Wagner [2] encode security properties as safety properties and use push-down automata and model checking to verify the absence of certain kinds of security problem.

General idea of static information-flow analysis dates back to Denning [4, 5]. See Sabelfeld and Myers' survey for an overview of this related work [10]. Most of these approaches rely on explicit code annotations and concentrate on sound enforcement of confidentiality properties. Here, we're concerned with integrity properties, and, since our analysis would have to be too conservative oth-

erwise, ABASH's approximations to information flows are unsound (largely due to the lack of precise specifications for information-flow in commands external to bash).

Other, more full-featured scripting languages like Perl and Ruby include forms of dynamic *taint checking* that can prevent unsafe arguments from reaching critical functions. A similar approach could be adopted by modifying the bash interpreter, but our experience suggests that there would be a large number of spurious warnings without providing some means of untainting inputs (as is provided automatically by Perl's pattern matching, for example).

As far as we are aware, no other research has attempted to statically analyze bash programs for security problems, but others have used proposed dynamic techniques for debugging bash scripts. The bash interpreter itself can be run with the `-x` flag set, which causes the interpreter to print a trace of each command's arguments before executing it. There is also an open source project called `bashdb`[10] that provides a gdb-style stepping debugger for bash scripts. Both of these approaches go some way towards making it easier to write correct bash scripts, but they don't provide explicit support for taint analysis or detecting the variable-expansion problems that ABASH does.

6. Discussion and conclusions

As noted in Section 4, ABASH is able to find bugs—many of which cannot be eliminated simply by running the scripts in a clean environment—in quite a few bash scripts distributed via popular free software repositories, without burdening the user with too many spurious warnings. It is not an entirely satisfying story, however; as noted in Sections 3.4 and 4.1, there are varieties of bugs that ABASH cannot reliably identify without also flagging far too many legitimate cases, and it is not at all obvious what might be done about this.

It is tempting to declare that these bugs—along with perhaps all of those that can be discovered with ABASH—can be avoided by following rather simple rules when writing scripts. In many cases this is true; putting a `-` or `--` before all non-constant arguments that are not intended to be options and quoting all non-constant arguments that are not intended to expand is a very good idea when when using bash. Unfortunately, this is not always possible; nesting of quotes, and especially of embedded commands[11] can make it difficult to reason about when quotation is required, and ruling out options in the remainder of the arguments is not possible with commands dependent on argument order.

It is also tempting to declare that the use of bare, unquoted strings in bash—and the ease at which they expand in such a wide variety of ways—is a mark of poor language design, but while these features do make bash difficult to analyze and bash scripts prone to error, the value of their convenience, especially when using the shell interactively, cannot be understated. Experience using the top-level loops of safer languages seems to confirm that such convenience is important; interaction with the system is much less convenient when every filename must be quoted and every string expansion must be made explicit.

We seem forced, then, to reexamine the basic idea of using exactly the same language scripting the system as for interacting with the system. There are clearly advantages for doing so—bash remains in use even given the widespread availability of scripting languages like Perl and Python—but could there be an unexplored middle ground that manages the best of both worlds? That is, could a single language have both an interactive variant and a scripting variant such that the interactive variant achieves the convenience of bash, the scripting variant is better suited to writing correct programs, and each variant feels natural to users used to the other? It seems that this could be achieved by choosing a scripting syntax that seems enough like a natural elaboration of the interactive syntax, but striking just the right balance is doubtless no easy task.

6.1 Future considerations

There are many corner cases to consider when analyzing bash, and ABASH does not yet cover them all. This lack of coverage is responsible for the shallow spurious warnings discussed in Section 4; additionally, we excluded from our testbed those scripts that could not be handled at all due to their reliance on unsupported features. Dealing with these problems, while sometimes an engineering challenge, is primarily a matter of committing the time to working out the details.

Beyond such concerns, there are several directions along which ABASH could be expanded.

Finding more bugs ABASH does not yet consider many sources of surprising behavior in bash scripts. For example, while some environment variables absolutely must be set to known values for a script to be secure—for example, the `$IFS` variable, which defines the characters to be considered as whitespace—others, like `$PATH`, are safe when under user control provided certain invariants are maintained. Giving ABASH the ability to track the current path—especially when this knowledge is not complete—would allow it to ascertain how much a script's behavior can be altered by the `$PATH` variable.

ABASH also does not handle many of the builtin commands that alter various aspects of bash's behavior. As mentioned in Section 3.3, many of the most frustrating of these builtins (from our standpoint) relate primarily to bash's interactive behavior, but there are still those that might conceivably be used within a script, and modeling them could allow for the detection of more potential bugs.

Limiting spurious warnings While the shallow warnings may simply be a matter to be overcome with sufficient programmer hours, the deep warnings generally stem from unknown invariants held by either the file system or external programs. It is easy to imagine augmenting the signature system to reflect this information, but this runs into several problems.

To begin with, describing the output behavior of a program may be tantamount to reimplementing it in many cases, so it is not obvious how descriptive of a language should be used for these output signatures. One would hope for the ability to specify `dirname`'s behavior to at least some degree, but `perl` is certainly out; what, then, of `sed`? It is also difficult to see how these signatures might be automatically generated, as the formatting conventions of `man` pages would not be there to assist us.

Describing invariants on trusted parts of the filesystem would run into similar problems, but there is also the issue of less trusted portions of the file system that may nevertheless be relied on by scripts. For example, a script may write a process ID to a file in `/tmp` then read it in later without performing any validation. Even ignoring the question of whether this is safe—discussed shortly—understanding the behavior being relied upon requires an abstract model of the filesystem and the effects external programs have upon it, raising the similar questions over the right level of descriptive power and the difficulty of automatic generation.

Finally, the particularly prevalent warning discussed in Section 4.1 cries out for more detailed analysis; if ABASH were able to better understand the degree of risk posed by potential option insertions, it could focus in on only the most dangerous cases and hopefully provide much a user with much better guidance. To accomplish this, we might extend our signatures to indicate whether

[10] `http://bashdb.sourceforge.net/`

[11] Bash allows the syntax `$(...)` as a less confusing alternative to `` `...` `` when it comes to the treatment of nested quotes, but many scripts continue to use the older syntax.

a certain command should be protected from externally provided options, or we could imagine specifying degrees of taintedness—with, for example, script parameters like `$1` ranking higher than the output of external commands—and warn only the most tainted of values might introduce an unexpected option.

We might also look for patterns in the use of particular variables (such as always being passed to a given command before any other arguments) or apply language-based heuristics (e.g., noting variables that contain the string `OPT`) in order to identify certain variables as permissible sources of options. This approach, while admittedly rather ad-hoc[12] does have the added advantage of applying equally well to the problem of determining whether the programmer meant or did not mean for a given variable always to expand to a single word.

Expanding* ABASH*'s scope ABASH currently looks for errors that may occur when bash scripts are run with input or in an environment other than what the script writer expected. Another possibility, especially when considering the filesystem, is that the environment is changing as the script is run. Indeed, it is well known that shell scripts are vulnerable to race conditions—for example, one may guard a sensitive file operation performed by a system maintenance script with `if [[ ! -L ]]`, intending that it will not be run on symbolic links, but a malicious user might still be able to replace a regular file with a link (perhaps to a file that the user could not normally access) between the execution of the test and the sensitive operation. Given the lack of atomicity in shell scripting, these attacks can in general be difficult to deal with.

However, unlike in the traditional programming setting, race conditions in shell scripts stem much less frequently from concurrency than they do from an environment shared by untrusted parties. Thus, given an abstract model of the filesystem, we can imagine extending ABASH with an understanding of trust relationships between users—as a simple example, it may be the case that `root` trusts no-one, other system users trust each other and `root`, and standard users trust only themselves and system users—allowing it to warn about potential race conditions involving untrusted parties. While reaching this level of understanding of bash would be no small task, it does not seem too burdensome in this case to ask for some amount of assistance from the programmer, as the benefit would likely outweigh the cost in a way that it does not for the sorts of errors ABASH currently looks for.

On a less ambitious note, ABASH currently has a very limited understanding of the argument conventions employed by external commands. Such conventions can sometimes be rather complex and subtle, and expanding ABASH's ability to understand the conventions of arbitrary commands that do not follow any particular standard—like the Unix `find` command—could allow it to discover bugs that might otherwise be particularly difficult to track down. One can also imagine combining such a detailed understanding of command arguments with some of the ideas discussed previously; although this would only compound the issues surrounding their realization, having a simple abstract model of the relation between input and output of standard Unix commands could allow ABASH to employ the something approximating the same reasoning as a human script writer. The ability to see a command as a simple unit with definite meaning and automatically reason accurately about several such commands strung together is precisely what we would hope for in an ideal analysis of shell scripts.

Acknowledgments The authors thank the anonymous reviewers for their excellent feedback on the submitted draft of this paper. We also thank the members of Penn's PL Club for their support and discussions. This research was sponsored in part by NSF Grants CNS-0346939 and CNS-0524059. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

[12] Several scripts in our testbed appear to have been written by German authors whose choice of variable names would naturally confound our natural choice of heuristics.

References

[1] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[2] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA, 2002. ACM Press.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[4] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[5] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[6] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[7] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006.

[8] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[9] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, Jan. 1999.

[10] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[11] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM Press.

[12] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium*, July 2006.