Saluki: Finding Taint-style Vulnerabilities with Static Property Checking

Ivan Gotovchits
Carnegie Mellon University
ivg@cmu.edu

Rijnard van Tonder
Carnegie Mellon University
rvt@cmu.edu

David Brumley
Carnegie Mellon University
dbrumley@cmu.edu

***Abstract*—We present Saluki, a new tool for checking taint-style (data dependence) security properties in binary code. Saluki provides a domain specific language for expressing taint-based policies. Saluki can find vulnerabilities in real programs for a number of CWE types, including those for command injection, weak PRNG seeds, and missing sanitization checks such as SQL escape routines or checks on buffer lengths. Saluki includes two new ideas in binary program analysis. First, Saluki uses μflux, a new static analysis technique for path-sensitive, context-sensitive recovery of data dependence facts in binaries. Second, Saluki introduces a sound logic system for reasoning over data dependence facts. We develop a domain-specific language on top of our logic system to express security properties as formal specifications. Saluki includes a decidable solver procedure to prove (based on the underlying logic) whether a set of data dependence facts satisfy a security property. Our evaluation shows that Saluki is capable of finding vulnerabilities in COTS x86, x86-64, and ARM software, including 0-days**

I. Introduction

Vendors continue to ship vulnerable programs to consumers. While source-code analysis holds promise for finding vulnerabilities, vendors often do not ship source code. As a result, we need techniques and tools for finding vulnerabilities in binary off-the-shelf software. Data dependency security properties, commonly called "taint vulnerabilities" [16, 25, 39, 40], are a crucial class of vulnerabilities. The Common Weakness Enumeration (CWE) database lists dozens of vulnerability types that boil down to tracing data dependencies and assuring tainted terms are either appropriately sanitized or not passed to critical functions.

Researchers have proposed a variety of dynamic techniques such as taint analysis to potentially detect such vulnerabilities. One set of techniques focuses on dynamic analysis that monitors execution flow, e.g., [8, 9, 11, 23, 31, 33, 36]. However, dynamic analysis requires runtime support and can achieve low coverage, both of which can result in missed vulnerabilities. On the other end of the spectrum is a static analysis, which promises to reason about entire functions or whole programs at once by abstracting program state [4]. As a result, static analysis tends to miss fewer problems, but can suffer high false positives if not done with care. Recent researchers have focused on symbolic execution, which blurs the line by using a mix of dynamic information to consider individual program paths [3, 13, 36, 37]. Symbolic execution, however, is still typically OS aware, requires significant runtime support, and relies heavily on extensive decision procedures to answer queries.

In this paper we present Saluki, a taint-style tool for statically checking security properties. At a high level, we define a property to be a predicate that holds over a set of one or more unique path executions. For example, one well-known security property is that an unsanitized `read(x)` should not flow into a `system(cmd)` call. In normal operation, Saluki first generates path- and context-sensitive data dependence relations on values that depend on `x`. Second, Saluki checks that a program, along with the data dependence facts, adheres to a security policy, e.g., whether `system(cmd)` depends on `x`. The security policy language is expressive enough to include CWE vulnerabilities such as SQL injection, command injection, and program-specific data flow security properties, and can also find vulnerabilities such as Heartbleed. Saluki works as follows:

> **Data dependency generation.** We propose *μflux* for generating data dependency facts in a path- and context-sensitive manner binary programs.
> **Policy specification.** The user provides a security policy, for example "`read(x)` should not flow into a `system(cmd)` call". We have developed a policy language that abstracts out binary specific details such as calling conventions, registers, etc. from the policy itself.
> **A logic engine** for checking security properties over *explicit* data dependence facts and *implicit* control dependence along paths.

A key idea in Saluki is μflux, which "executes" parts of a program to find data dependencies. μflux can start executing at *any* program instruction within an emulator. The emulator catches all reads and writes to registers and memory before they happen. The first time a program term is used, μflux initializes it via a *seed value* policy, such as choosing a random value or a constant. μflux then follows execution, registering definitions and uses of each program term (e.g., register and memory location) in a database. Like a fuzzer, μflux can re-execute paths with new seed values, with the advantage that μflux can start at any instruction.

μflux emulation has two path selection modes: deterministic mode, which evaluates branches based on the emulator state,

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23xxx
www.ndss-symposium.org

and non-deterministic emulation that follows all program paths regardless of the branch condition. Note we do not claim executing from any address in an emulator is novel (e.g., similar concepts are proposed in [20, 38]), but instead *customizing on-the-fly software-based execution to identify path- and context-sensitive data dependencies*. Compared to static analysis, μflux allows us to side-step issues such as variable recovery, alias analysis, and even object-oriented code identification during binary analysis, and yet still collects high-fidelity facts about data dependencies. Compared to dynamic analysis, μflux does not require runtime support, and can start in arbitrary program locations to produce path- and context-sensitive information. Given the above new ideas, we ask three research questions:

1. Can we express real security policies over real programs in Saluki that finds real bugs? We show a variety of CWE's (Common Weakness Enumerations) can be formally modeled in Saluki, and that these specifications find real bugs in binary programs. In particular, our experiments use Saluki to (1) find known vulnerabilities (e.g., Heartbleed) and (2) find 6 new zero-day vulnerabilities in COTS SOHO router software, including 1 in `Lighttpd`, a high performance webserver used in thousands of SOHO products and over 400K internet-facing sites (including `xkcd.com`).[1]

2. Is the proposed μflux path- and context-sensitive data dependency approach scalable to complete programs? For example, related approaches like micro execution [20] were only demonstrated on individual functions, and flood execution [38] on portions of malware samples. More generally, path- and context-sensitive enumeration in static analysis is sometimes expensive. We find μflux hits a sweet spot by combining ideas from micro execution and flood execution. Like dynamic analysis, we can collect path- and context-sensitive information. However, μflux does not need OS or runtime support (e.g., we analyze ARM binaries on x86 machines), is not limited to start at the beginning of a program path, or restricted to paths with a known input like dynamic analysis. μflux is similar to static analysis in that it achieves high coverage, but at the same time side-steps issues such as developing alias analysis, variable recovery, and type recovery in order to get meaningful results.

3. Can we develop an decision procedure checking security policies written in our language and logic over data dependencies? We develop a decision procedure for policies over data dependency facts, and show that it scales to programs of moderate size and per-path data-flow facts. Since Saluki uses a constructive proof system to verify properties over a program model, we get information not just on whether the policy is violated, but on how it is violated (e.g., an example program path that violates the policy). Our formalization of the program model finds particular application in COTS binaries, but is also sufficiently general to model higher-level languages.

Our overall contributions are the Saluki tools and techniques, including:

1) A new approach for collecting path- and context-sensitive data dependency information called *μflux*.
2) A verification framework comprising (a) a sound logic system, (b) a security policy language, and (c) a solver decision procedure. The framework enables efficient, decidable detection of common taint-style (data dependence) vulnerability patterns.
3) An experimental evaluation where we use Saluki for vulnerability discovery. We demonstrate vulnerability specifications, discovering 6 zero-day vulnerabilities in five different vendor SOHO router products across 5 different CWEs. We also use Saluki to find known vulnerabilities in the Linux kernel, OpenSSL (Heartbleed), Pidgin, and C++ compiled binaries.
4) An open source implementation. Saluki currently supports the ARM, x86, and x86-64 architectures, and can parse ELF, PE, and in general any executable format that IDA Pro can parse.[2] It is available at https://github.com/BinaryAnalysisPlatform/bap-plugins/tree/master/saluki.

II. Modeling and Checking Security Properties

In this section we give an operations-based overview of Saluki, including the types of vulnerabilities we target. In the following sections we cover the formal semantics.

A. Saluki Operation

Users check security policies in two steps. First, the user specifies their security policy in the Saluki policy language. Saluki security policies describe security taint-style path properties. Policies have two parts: (a) identify Program patterns of interest, e.g., API calls like `recv` and `system`, and (b) the data dependency relationships to check between locations of interest. Our design has three goals:

1) Extract out data dependencies with high fidelity, low false positives, and over many paths. This goal is necessary because we want data dependencies to be real dependencies (i.e., "must" not "may" dependencies), and to reason about as much as the program as possible. Dynamic analysis gives high fidelity (e.g., we know concrete values for registers), has no false positives (every data dependency in the trace is a true dependency), but is over only one path. Static analysis for proving programs are bug-free (i.e., safe) can reason over an entire routine or program, but often have high false positives (e.g., many data dependencies extracted are not true dependencies), especially at the binary level. Saluki extracts out all data dependency flows (up to an execution depth) by "executing" snippets of code, and extracting out all dependencies along the executed snippet. As such, it is a type of path and context sensitive analysis over many paths, and has similar fidelity to dynamic analysis since each execution is a witness for a data dependency.
2) Reason about security policies over all extracted data dependency flows at once. In particular, we do not want the explosion of reasoning over each path separately, as many paths share similar dependencies. In Saluki we use a custom logic to reason over all extracted data dependencies in a sound manner.
3) Do not litter the policy language with low-level details like calling conventions, memory cells, etc. when possible.

[1] http://trends.builtwith.com/web-server

[2] Saluki does not require IDA Pro (e.g., it works without IDA on ELF files), but benefits from it when present by using it as a plugin for parsing executable container formats and for advanced symbol recovery.

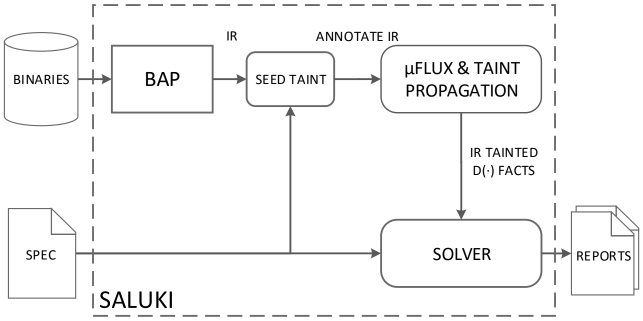

Fig. 1: Saluki Architecture

Saluki uses a lightweight policy langauge with a gear towards pattern matching and abstracting out low-level details from the policy itself. The policy abstracts low-level details such as whether `read` puts the results on the stack (x86), or a register (as in ARM).

Put together, the overall Saluki architecture is shown in Figure 1 (Appendix E). When run, Saluki:

a) Loads in the specification.
b) Parses the binary into an intermediate representation (IR) suitable for analysis.
c) Runs μflux to collect data flow facts about executions from every specified source.
d) Runs a solver over the policies, program, and collected facts. The solver determines whether the property holds or not. The solver implements a novel logic system and algorithm built around the the specification DSL.
e) Saluki outputs example paths where the property does not hold. The actual output is not a full path, but instead a condensed form of the tainted instructions and a flow ID (outputting a full path is possible, but likely very large).

A Running Example: The Saluki policy language is designed to check *explicit* taint-style data dependency properties that have *implicit* control dependencies. We observe that many Common Weakness Enumeration (CWE) specifications fall into this category, including CWE-89 (SQL Injection), CWE-337 (Predictable Seed in PRNG), and others.

For concreteness, we use CWE-78: Command Injection Attacks (Rank 2 in the CWE database) to explain our specification language and as a running example in the rest of the paper. Command injection vulnerabilities arise when input flows from an input source to a sink function that executes code. For example, consider a specification that says `system` should never use data from `recv`. The Saluki specification is:

```
prop recv_to_system ::=
  recv(_,*buf,_,_), system(*cmd) |- never
  s.t. cmd/buf
```

Listing 1: Network input should not reach `system`

The specification in Listing 1 defines a *security property*, which specifies two common APIs used for command injection.

Within a property the user specifies locations of interest using a *pattern*. Patterns can match (a) with program terms that load, store, use or define values or (b) with program terms that perform control flow, such as conditional branches or function calls. For usability, we let users specify taint start points of API calls using a C-like syntax with positional arguments. At a low level Saluki parses header files, identifies functions starts and calculates argument expressions (e.g., registers or stack offsets) based on the detected ABI and calling convention. Saluki patterns are extensible: a user can create their own patterns and upload their own header files without changing Saluki.

In Listing 1, the user has specified two APIs of interest: `recv`, which takes four arguments, one of which is a pointer to a variable called `buf`, and `system`, which takes one pointer argument called `cmd`. We use the same notation for variable names, and that the `'*'` (asterisk) denotes a pointer. For usability, we allow users to write `'_'` (underscore) for locations not of interest.

Property bodies consist of sequents of the form $\mathrm{p_1, \dots, p_n \mid\!- q_1, \dots, q_m}$ where $\mathrm{p_j \mid j \in 1, \dots, n}$ and $\mathrm{q_i \mid i \in 1, \dots, m}$ are patterns denoting premises and conclusions, respectively. The user refines sequents to include data dependencies of the form $\mathrm{q_i/p_j}$, read $\mathrm{q_i}$ is data-dependent on $\mathrm{p_j}$. Note that we may also specify data dependence between premises alone, as in Listing 1. In the above case the rule introduces a flow *constraint* `cmd/buf`, meaning the entire rule represents only the pairs of calls to `recv` and `system` where `buf` flows into `cmd`. `never` corresponds to a logic construct that forms a positive assertion in the specification. Note that specifications must be supplied as positive assertions to detect violations, as the downstream proof system is *constructive*.

Binary Processing: Saluki takes as input a specification and binary to check. Saluki first lifts instructions to an intermediate representation (IR) using BAP [1, 10], an open-source plugin-based binary analysis framework. The IR provides a precise semantic for each instruction, along with meta-information such as linking particular addresses to symbol names and function starts (e.g., as determined by Byteweight [7] or IDA Pro), as well as matching up terms from usability features such as annotating program locations with respect to a known header file. The annotations are represented in the IR as a synthetic `arg` term. We call each syntactic IR element a *term*. Each term has a unique label, which can ultimately be tied back to the instruction that generated it. Note that a single instruction may correspond to several IR terms, and therefore several labels may map back to the same instruction.

Taint Seeding: Saluki analyzes the specification for variables used in constraints. Each constraint variable is linked to a program location, which is then marked in the IR as a taint seed. In our running example, `cmd` is a constraint variable used in `recv`, causing Saluki to identify the proper memory location corresponding to the `cmd` argument in all terms named `recv`. As is customary, Saluki uses unique identifiers to identify each taint seed.

μflux: Saluki then uses μflux to collect data dependency facts. μflux is implemented as a custom interpreter that executes the IR of the binary. This design decision allows us to implement Saluki on every platform supported by BAP, including ARM, x86, or x86-64. The Saluki μflux interpreter looks for taint seeds, and starts executing with respect to a seed policy. By default, Saluki uses a random seed policy, where each initial reference of a variable is assigned a value uniformly at

random. Saluki also takes in a parameter on whether to execute branches based on (1) the running interpreter state, which we call *deterministic branch evaluation*, or (2) "flood" execution, which follows both sides of the branch regardless of evaluation, which we call *non-deterministic branch evaluation*. By default, Saluki uses a non-deterministic branch evaluation.

During execution the interpreter (1) propagates taint as needed according to specified taint semantics (see Appendix A), and (2) generates def-use facts of the form $\mathcal{D}(l', R', l, R)$ for tainted terms, meaning the IR variable R' at location l' depends upon the value R defined at location l. Note that although we generate these facts using μflux, the underlying logic representation is general enough to handle facts generated from other analysis techniques. μflux keeps a separate execution state on each path, generating path and context-sensitive dependency facts. μflux executes through instructions, including calls, returns, and other statement types often problematic for static analysis. μflux stops executing on a path when any of the following conditions are met: (a) Saluki hits a pre-defined maximum number of instructions to execute, similar to bounded model checking (the default is 10 million basic blocks of instructions), (b) Saluki hits a call to a function that is not modeled (e.g., library functions that are dynamically linked at runtime and not in the binary itself), or (c) Saluki hits a jump with an indirect target.

The Saluki Solver: Saluki takes in the program IR, the data flow facts, and a security policy, and tries to prove all properties specified in the policy. The properties have the form $p \vdash p'$ $\mathbf{s.t.}$ c. The patterns in p and p' specify locations of interest subject to data flow constraints c. Saluki's goal is to prove the property, or find counterexamples where the property does not hold. When Saluki μflux is run using deterministic branch execution, Saluki is fully path- and context- sensitive. When Saluki μflux runs in non-deterministic mode, Saluki merges taint information on a per-path basis, making it context- and flow-sensitive.

Since properties are of the form $p \rightarrow p'$, Saluki tries to prove a counter-example $\neg(p \rightarrow p') \Leftrightarrow p \wedge \neg p'$. A counter-example is a set of program terms and generated data dependencies that serve as a witness that the policy is violated. As a result, Saluki is *constructive:* it doesn't just say there is a violation, but gives the specific path and data dependencies used to show the property can be violated. Note a design-interplay between the logic system and specification. Saluki internally reasons about negation, but in a constructive sense. The Saluki DSL, however, does *not* allow a user to specify negation, as it would increase the computational complexity of the overall logic system (possibly increasing the complexity class).

B. Vulnerability Specifications

Taint Vulnerabilities: The CWE-78 specification is a simple yet representative of a general class of CWE specifications of flows from sources to sinks, including:

CWE-89 SQL Injection SQL injection vulnerabilities commonly arise when user input is used as part of SQL statement. For example, suppose that an API offered two primitives: `a = sqlcreate(s)`, which creates a query handle `a` for a string `s`, and a `b=sqlsanitize(r)` function that escapes all SQL metacharacters in a string `r` escapes any SQL characters. The Saluki specification for this CWE would be the same as command injection up to API renaming.

CWE-337/676 Predictable Seed in PRNG We translate this CWE into checks between known insecure sources of randomness, and known PRNG sinks. For example, the Saluki specification for sources `p = time()` flowing into sinks `srand(q)` is almost the same as the above SQL and OS property. A program containing both calls is vulnerable along paths where `q` ever depends on `p`, and is safe if no such path exists.

CWE-252 Unchecked Return Values Saluki can also check that tainted values are used in checks, modeled as control flow jumps based on tainted values. The Saluki language includes `when c jmp x`, read as when condition `c` is satisfied, jump to location `x` (which can be either a computed or immediate address).

In particular, consider CWE-690: "Unchecked Return Value to NULL Pointer Dereference." In practice, this translates to checking that the return value of memory management functions like `calloc` and `malloc` are checked against NULL. For example, Listing 2 shows source from `Lighttpd`'s `stat_cache.c`. It turns out that there's no check on the return value `keys` after the `calloc` call. Note that the variable `sc->files->size` is under attacker control.

```
keys = calloc(1, sizeof(int) * sc->files->size);
```

Listing 2: Unchecked `calloc`

To detect this flaw, one needs to identify (a) `calloc` call sites in the program, and (b) assert that the return value `keys` is always checked. The Saluki specification for `calloc` is:

```
prop calloc_maybe_checked ::=
  p := calloc(_) |- when c jmp _ s.t. c/p
```

Listing 3: Unchecked `calloc`

Listing 3 matches all cases where some jump condition `c` depends on the return value of `calloc`. In the `Lighttpd` specification, we first provide a pattern for matching `calloc` call sites, and bind a return value to a symbolic variable `p`.

Second, we provide a pattern for matching *control flow* conditional statements, and bind the condition expression to a symbolic variable `c`. Using these variables, we can describe the cases where some conditional expression `c` uses the return value `p`. We express this by `s.t. c/p`, which reads "such that `c` is data dependent on `p`". Interesting cases occur where *no* conditional statement uses (or depends on) a `calloc` return value. The negation of the property flags instances where Saluki could not satisfy this assertion.

Saluki's output produces 75 instances of unchecked `calloc` calls in the ARM-compiled `Lighttpd` binary we considered. Each instance constitutes a legitimate flaw in `Lighttpd`, but not necessarily a vulnerability. Our discovery has two effects: (a) a cursory inspection of the 75 cases revealed one security-critical operation which may cause a memory corruption, culminating in a new CVE, and (b) the maintainers of `Lighttpd` introduced `calloc` checks for over 40 call sites.[3] This mitigates the potential of additional vulnerabilities and prevents denial-of-service attacks due to unchecked NULL pointers that may result from out-of-memory errors. Similarly, Saluki can be used to detect CWE-252, "Unchecked return values" which captures the general class of vulnerabilities in spirit of the above.

Templates: Many of the CWEs share a common structure of "check this after that", and only logically differ in the specific APIs checked. Saluki provides a set of reusable template rules that only requires identifying the relevant APIs, and does not require repeating the entire rule.

Limitations: Saluki does not specifically reason about memory corruption vulnerabilities such as buffer overflows. In particular, such vulnerabilities often require counting, e.g., does the source byte count exceed the destination byte count. We leave a more thorough extension to such vulnerabilities as future work. Saluki, however, can check common API idioms that may be insecure. For example, while we cannot check input size directly, we can check that user inputs do not flow to known unsafe functions, e.g., for strcpy:

```
prop if_strcpy_dst_depends ::=
  recv(_,*p,_,_),strcpy(_,*q) |- never
  s.t. q/p
```

Listing 4: Network input reaches strcpy

We demonstrate this capability by rediscovering the Heartbleed vulnerability.

III. MICROFLUX

A. Why Microflux?

An integral part of Saluki is deciding whether the expressed security policy is satisfied by a program model, where the program model includes a set of data flow facts. Unfortunately, accurate inter-procedural, context-sensitive analysis data flow analysis that reasons about aliasing is undecidable [27]. Therefore, any analysis seeking to handle such programs must approximate.

One approach is to statically enumerate paths, e.g., similar in spirit to the source-based techniques from PQL [29] and Property Graphs [39]. Static enumeration typically does not consider branch predicates, and thus may generate infeasible paths. A second approach is to consider only realizable paths, e.g., by enumerating concrete inputs, using fuzzing or symbolic execution.

μflux strikes different design points in this space. First, μflux simulates concrete dynamic behavior. This allows us to sidestep expensive symbolic reasoning techniques in static analysis (e.g., aliasing) and binary analysis (e.g., variable recovery, object-oriented code identification). Second, execution behavior is parameterizable by a number of policies. This overcomes limitations of dynamic techniques such as fuzzing and concolic execution which (a) can only detect bugs triggered by an executed path at runtime, and (b) must always start at the program entry point.

B. Microflux Design

Branching Policy: Recall μflux can evaluate branches either deterministically (i.e., by evaluating the branch in the specific execution context) or non-deterministically (by evaluating all branch points). When operating deterministically, we have a witness of initial state that proves a particular branch is feasible. This avoids exploring impossible branches, e.g., due to conditions of the form if(some big expression that is false).

When operating non-deterministically, we explore paths regardless of the branch predicate. This allows Saluki to explore a larger portion of the program, but at the cost of potentially exploring infeasible paths. We note that this problem is not specific to μflux, and occurs in other static analysis techniques, e.g., [39]. In our evaluation, we have experimented with both, and set the default Saluki policy to non-deterministic mode. Our experiments show that non-deterministic mode finds more vulnerabilities, and that infeasible paths do not create overwhelming false positive policy violations. However, users can configure which policy to choose based upon their own experience.

Execute Anywhere: μflux can start executing from any location in the intermediate representation. This feature allows us to start executing at program points of interest *with respect to the specification*, removing the need to initiate execution at program entry. In practice we find that ignoring contextual information before a program point of interest does not impact the correctness of our results. In fact, for our examples in §II it is preferable to start tracking specific information at its creation point (e.g., precisely when a return value is defined, or input from recv occurs).

Data Dependence: Initial register and memory state can be defined by a custom seed policy (e.g., populated by concrete values). In this paper, we use μflux with a random seed policy. Each initial read of a register or address is assigned a value uniformly at random. This is a pragmatic choice; we are interested in the semantics of data flow over program terms (the transfer of values) to find vulnerabilities, not the values themselves. Data dependence facts are extracted after propagating taint over terms in the IR according to the operational semantics in Appendix A. The full grammar and operational semantics for the BAP IR is available and omitted for brevity.[4]

Note the distinction of a branching policy for fact generation and a specification in Saluki. Saluki quantifies over *explicit* data flow relationships in the specification. μflux is the technique used to supply Saluki with data dependence facts. We use explicit data flow (e.g., [17, 32]); we leave considering implicit flows as future work. Saluki does not directly consider control flow or control dependence used to generate data flow facts: these are *implicit* attributes. The branching policy gives us the flexibility to parameterize *how* μflux generates facts with respect to these implicit attributes. This is an intentional design decision: data dependence facts on their own are sufficient to express many vulnerabilities, as in §II. By delegating fact generation to μflux, we can experiment with different tradeoffs (or even use other techniques like fuzzing). We note that explicit

[3] Commit 566cf.

[4] https://github.com/BinaryAnalysisPlatform/bil

consideration of implicit data flows is an interesting opportunity which we leave as future work.

Precision and Tradeoffs: Recall our first design goal: to extract data dependencies with high fidelity, low false positives, and over many paths. μflux finds "must" data dependencies, but may explore infeasible paths by ignoring branch conditions. This is one potential source of unsoundness. Our results show that the tradeoff in exploring infeasible paths is is acceptable, and we observe low false positives. Two reasons contribute to this: (a) data dependence facts that do not affect the terms in a specification play no role in proving a specification; i.e., parametricity of specifications constrain opportunities for unsound results, and (b) most of the vulnerability specifications (e.g., injection), intend to prove the *absence* of data dependence relationships, including and in spite of unrealizable paths.

Recall our second design goal: to reason about security policies over all extracted data dependency flows at once. In deterministic mode, the data dependence facts of a single path are used to prove a specification. In non-deterministic mode, facts are gathered along multiple paths. Extraction of data dependence is always performed in a path- and context-sensitive manner. However, in non-deterministic mode, data dependence facts are merged after completing each path exploration, resulting in a flow-sensitive analysis aggregated over individual paths. We make this precision tradeoff so that we can run a constructive proof over all data dependence facts at once. Our results indicate that this tradeoff finds vulnerabilities without needing to repeatedly reason over duplicate data dependence facts for multiple paths individually. Note that in non-deterministic mode, Saluki still supplies the program terms responsible for violating a specification (so that a path witness can be reconstructed).

We note that the μflux emulation engine may have implementation errors, causing additional sources of unsoundness. We have tested emulation across a suite of Coreutils by bi-simulating Saluki's engines and a native x86 CPU. In our experiments, emulation is 99.99% accurate for handled instructions, and thus we do not consider this a serious issue.[5] We stress that the above design choices and tradeoffs are particular to μflux, and do not affect the soundness of the prover. We developed our formalism to achieve a *reusable*, sound reasoning engine over data dependence facts, thereby isolating fact generation (and potential sources of unsoundness) to a particular technique.

IV. Saluki Logic System and Language

The Saluki Logic System and Language enables sound and decidable reasoning over explicit data dependence facts. We introduce a security policy language to express vulnerability patterns as specifications. A constructive proof procedure checks whether a program violates a property. Note that the logic system is general and may be reused with dynamic or static analysis techniques that can generate data dependence facts.

Syntax and Semantics

To define properties we use a domain specific language. The language grammar allows us to specify a property as a sequence of patterns and a set of constraints. The property holds if all patterns match under the given constraints. The grammar of the language is shown in Figure 3. We formally define Saluki semantics as a set of axioms, shown in Figure 2.

$$
\begin{array}{lll}
r & ::= \mathbf{prop}\, id \triangleq p \vdash p' \,\mathbf{s.t.}\, c & :: \textbf{property} \\
p & ::= & :: \textbf{patterns} \\
 & \mid p1, \dots, p_m & :: \textbf{list} \\
 & \mid v := id(v_1, \dots, v_m) & :: \textbf{sub} \\
 & \mid v := v' & :: \textbf{def} \\
 & \mid \mathbf{when}\, v\, \mathbf{jmp}\, v' & :: \textbf{jmp} \\
 & \mid \mathbf{never} & :: \textbf{bot} \\
v & ::= id \mid *id & :: \textbf{values} \\
c & ::= c_1, \dots, c_m \mid v'/v \mid \mathcal{P}(v) & :: \textbf{constraints}
\end{array}
$$

Fig. 3: The Saluki Specification Language

Definition 1. The *abstract program model* is a triple of propositional functions $\mathbb{P} = (\mathcal{T}, \mathcal{D}, \mathcal{P})$. The proposition $\mathcal{T}t$ denotes that a term t exists. The proposition $\mathcal{D}(l', R', l, R)$ denotes information flow (data dependence) from a variable R *defined* in a program term with label l to a variable R' *used* in a program term with label l'. The proposition $\mathcal{P}(p, l, R)$ denotes a user-defined predicate p that holds for a variable R used in a program term with label l.

Definition 2. A *program term* t is an ordered 5-tuple $(\mathsf{L}_t, \mathsf{S}_t, \mathsf{C}_t, \mathsf{D}_t, \mathsf{U}_t)$ where

L_t is a Label that uniquely identifies a term,
S_t is a set of static Successors,
C_t is a set of program variables that affects the Choice of a successor,
D_t is a set of program variables that are Defined by a term, and
U_t is a set of program variables Used in the term.

We now gently introduce all parts of Saluki language, from top to bottom.

Property: The axiom (prop) states that if a property $p \vdash p' \,\mathbf{s.t.}\, c$ holds then the match of a pattern p under a constraint c must imply matching of a concatenation p, p' under the same constraint.

Patterns: Patterns are user-specified expressions in our syntax. Patterns contain logical variables. Logical variables are bound to variables in the body of a matched program term The process of binding logical variables to program variables is called a *valuation of a pattern* and is denoted with a proposition $[\![p]\!]_v \overset{t}{=} R$ that is inductively defined for each pattern. The proposition $[\![p]\!]_v \overset{t}{=} R$ states that a pattern p valuates a variable v to R in term t.

Pattern $\mathbf{never}$ never valuates. Axiom (never) states that every attempt to valuate $\mathbf{never}$ is absurd.

Pattern $v := v'$ matches with any definition term t. This pattern may bind a logic variable v to some program variable from a set D_t and bind v' to some program variable from a set U_t. The semantics are given by axioms (def-v) and (def-v') respectively.

Pattern $\mathbf{when}\, v\, \mathbf{jmp}\, v'$ matches with a jump term t. This pattern valuates a variable v to some program variable from a set C_t and variable v' to some program variable from a

[5] https://github.com/BinaryAnalysisPlatform/bap-veri

$$\llbracket \mathbf{never} \rrbracket_v \overset{t}{\doteq} R \rightarrow \bot \ \texttt{(NEVER)} \qquad \llbracket v_0 \coloneqq id(v_1, \ldots, v_m) \rrbracket_{v_i} \overset{t}{\doteq} R \rightarrow \mathcal{T}t \wedge \arg(id, i) = R \ \texttt{(SUB-A)} \qquad \llbracket v \coloneqq v' \rrbracket_v \overset{t}{\doteq} R \rightarrow \mathcal{T}t \wedge R \in \mathsf{D}_t \ \texttt{(DEF-V)}$$

$$\llbracket v \coloneqq v' \rrbracket_{v'} \overset{t}{\doteq} R \rightarrow \mathcal{T}t \wedge R \in \mathsf{U}_t \ \texttt{(DEF-V')} \qquad \llbracket \mathbf{when}\, v\, \mathbf{jmp}\, v' \rrbracket_v \overset{t}{\doteq} R \rightarrow \mathcal{T}t \wedge R \in \mathsf{C}_t \ \texttt{(JMP-V)} \qquad \llbracket \mathbf{when}\, v\, \mathbf{jmp}\, v' \rrbracket_{v'} \overset{t}{\doteq} R \rightarrow \mathcal{T}t \wedge R \in \mathsf{S}_t \ \texttt{(JMP-V')}$$

$$p\, \mathbf{s.t.}\, v'/v \rightarrow v \in \mathcal{V}(p) \wedge v' \in \mathcal{V}(p) \rightarrow \llbracket p \rrbracket_v \overset{t}{\doteq} R \wedge \llbracket p \rrbracket_v \overset{t'}{\doteq} R' \wedge \mathcal{D}(\mathsf{L}_{t'}, R', \mathsf{L}_t, R) \ ^{\texttt{(DEP)}} \quad p\, \mathbf{s.t.}\, \mathcal{P}(v) \rightarrow v \in \mathcal{V}(p) \rightarrow \mathcal{T}t \wedge \llbracket p \rrbracket_v \overset{t}{\doteq} R \wedge \mathcal{P}(p, \mathsf{L}_t, R) \ \texttt{(PRE)}$$

$$\llbracket p, p' \rrbracket_v = R \rightarrow \llbracket p \rrbracket_v = R \vee \llbracket p' \rrbracket_v = R \ \texttt{(COMMA-P)} \quad p\, \mathbf{s.t.}\, c, c' \rightarrow p\, \mathbf{s.t.}\, c \wedge p\, \mathbf{s.t.}\, c' \ \texttt{(COMMA-C)} \quad p \vdash p'\, \mathbf{s.t.}\, c \rightarrow p\, \mathbf{s.t.}\, c \rightarrow p, p'\, \mathbf{s.t.}\, c \ \texttt{(PROP)}$$

Fig. 2: The Saluki Language Semantics

set S_t. This semantics are given to by axioms (jmp-v) and (jmp-v') respectively.

Pattern $v_0 \coloneqq id(v_1, \ldots, v_m)$ matches with function call term t which calls a function named id. Logic variables are valuated positionally, relying on a function $\arg(id, i)$ that valuates the i-th argument of function id according to a used calling convention.

The axiom (comma-p) inductively extends the above axioms to a list of patterns.

Constraints: Patterns are matched under some (possibly empty) set of user-specified constraints. The match is defined as an inductive proposition for two possible constraints. (1) If a pattern p matches under constraint v'/v it creates the implication that if both v' and v occurs in the pattern p, then they must be valuated to such program variables for which $\mathcal{D}$ holds. (2) If a pattern p matches under a user-defined predicate constraint $\mathcal{P}(v)$ it creates the implication that if a logic variable v occurs in the pattern p then it should be valuated to such program variable for which $\mathcal{P}$ holds.

An axiom (comma-c) denotes that a pattern p matches under a list of constraints if it matches with each constraint.

We present details of the Saluki Proof System in Appendix D.

V. Implementation

Saluki is implemented in 1,675 lines of OCaml code. It is built on top of the BAP platform, which currently supports lifting of the x86, x86-64, and ARM architectures to an intermediate representation. BAP performs CFG recovery and inference of function prototypes for the GNU C Library. BAP uses LLVM as a disassembly backend. Saluki is released online [6] in support of open science.

VI. Evaluation

We evaluate Saluki with respect to the following research questions:

1) What classes of real-world vulnerabilities can we detect using Saluki? We tested 5 unique security policies for detecting taint-style vulnerabilities, and found 6 zero-days in COTS binaries (§VI-C). We also evaluate Saluki on known vulnerabilities found in previous work that detects taint-style vulnerabilities in source code [39, 40]. We find that we discover strictly more vulnerabilities by emulating μflux in non-deterministic mode.
2) How fast is our implementation, and what coverage do we achieve?

We perform a pure performance benchmark to test the speed of μflux using an empty security policy in non-deterministic execution mode. Our implementation takes 54 seconds on average per binary, averaged over all binaries in the Coreutils [2] test suite. We measure the number of IR instructions evaluated per second using μflux. At a high-level, we find that we are competitive with binary instrumentation-based micro execution [20]. We achieve 96% statement coverage, averaged over the Coreutils test suite with an empty security policy (§VI-G).

A. Experimental Setup

Our experiments were performed on an Ubuntu 14.04 LTS virtual machine with an Intel i7 2.2GHz CPU core and 6GB RAM.

We picked five binaries on the attack surface of five active, distinct vendor SOHO routers for vulnerability discovery. Binaries are considered to be on the attack surface if network input is processed by the binary.[7] Our analysis considers binaries extracted from the SOHO router firmware of Cisco, Linksys, Belkin, Airlink, and Buffalo. We also evaluated Saluki on known vulnerabilities in the Linux kernel, OpenSSL (Heartbleed), Pidgin, and C++ compiled binaries. With the exception of the C++ binaries, our choice is based on previous work that detects taint-style vulnerabilities in source code [39, 40]. We chose C++ binaries to demonstrate the additional novel capabilities that Saluki offers.

For performance measurement we used the set of ARM Coreutils binaries comprising 100 binaries. We evaluate the speed and coverage of μflux on Coreutils due to its wide use, reflection of real programs, and popularity for benchmarking.

B. Threat Model

Vulnerabilities differ from traditional software flaws in constituting a security risk. Additionally, what constitutes a security risk is highly context-dependent. One example is CWE-337, where a PRNG is initialized from a predictable seed, such as a process ID or system time. In our results we found two binaries that seed `srand` with time: one uses it for generating web session cookies, while the other uses it for randomizing an update schedule. The former presents a clear security risk, while the impact of the latter is unclear, and likely does not pose a security risk. Saluki finds specification violations which may or may not be exploitable. However, all of Saluki's reports in our tests correspond to insecure programming practices.

[6] https://github.com/BinaryAnalysisPlatform/bap-plugins/tree/master/saluki

[7] Automatically determining binaries on the attack surface of a device is an interesting point of future work, but outside of the scope of this paper.

Binary	Vuln ID	CWE	Flaws	Time (s)		Facts		Insns (K)		Cov (%)		Fns	Spec
			(0-days)	det	fl.	det	fl.	det	fl.	det	fl.		
Lighttpd	CVE-2016-1545	252	75 (1)	33	52	962	1,919	330.0	2,724.1	22	99	99	C4
admin.cgi	CVE-2016-1334	20/78	10 (1)	45	108	122	1,448	656.3	22,894.9	31	96	141	C3
admin.cgi	CSCuy68380	337	1 (1)	41	43	37	48	3.7	144.1	26	100	3	C5
pathload2	VU #911048	120	1 (1)	11	93	0	17	8.0	16,086.0	13	100	10	C1
easyconf	-	78	1 (1)	9	11	0	30	8.1	139.1	19	100	2	C3
cm.cgi	-	120	1 (1)	9	10	0	74	1.6	30.0	9	100	1	C2
openssl	CVE-2014-0160	120	5	228	301	23,999	25,817	61.8	88.7	15	20	972	C9
ozwpan.ko	CVE-2013-4513	120	1	70	72	160	608	0.3	16.7	17	40	2	C9
libmsn.so	CVE-2013-6482	20/676	1	44	46	48	5,032	15.0	119.1	12	54	18	C7
watchstatus.cgi	CVE-2015-6910	89	4	44	46	132	784	24.1	321.3	20	82	27	C6
audiotrack.cgi	CVE-2015-6911												

TABLE I: Saluki Analysis Results for 0-days (`Lighttpd` through `cm.cgi`) and and known vulnerabilities (`openssl` through `audiotrack.cgi`). **Fns** is the number of functions. Deterministic mode is denoted by **det**, and non-deterministic (flood) mode by **fl.** Shaded binaries indicate the vulnerability could only be found with *flood* execution mode (i.e., flood mode is better). **Cov** gives the percentage coverage of IR instructions for deterministic and flood mode, respectively.

C. Zero-day Vulnerability Discoveries

Table I enumerates our discovery of 6 zero-days in five distinct COTS vendor products. We distinguish between insecure programming practices (flaws) and confirmed vulnerabilities (zero-days), both of which are automatically reported by Saluki. Notably, our discoveries range over a variety of CWE classifications, illustrating Saluki's application to detecting a diverse set of flaws.

1) admin.cgi: The `admin.cgi` binary exposes device administration through a webpage. Saluki found two distinct vulnerabilities in `admin.cgi`. The first vulnerability we found in `admin.cgi` is a command injection to `system`. Saluki flagged the code location in the binary where the result of a `sprintf` call flows to system. This vulnerability allows an unauthenticated attacker to execute remote code through the web-interface, and was assigned CVE-2016-1334. The second vulnerability in `admin.cgi` uses a random value to generate a web session cookie. However, the PRNG is seeded by a predictable value: `time(NULL)`, allowing an attacker to guess the stream of random number generation with a smaller key space of initial seed values than brute force.

2) pathload2: The `pathload2` binary is used to determine bandwidth of an Internet connection. Saluki reports a direct data flow from the `recv` buffer to the source buffer of `strcpy`. The `strcpy` does not validate the length of the source string, resulting in a buffer overflow. This vulnerability was acknowledged by CERT, but required further coordination with the original software author for confirmation and resolution. We were unable to contact the author after multiple attempts.

3) easyconf: The `easyconf` binary is a third-party binary that to enables remote configuration of the router. It contains a command injection vulnerability similar to that found in `admin.cgi`. We discovered it using the same specification, by detecting a flow from `sprintf` to `system` through an attacker-controllable string.

4) cm.cgi: The `cm.cgi` allows remote configuration of wireless network settings. A common convention for CGI binaries in embedded devices is to read input through environment variables. A buffer overflow vulnerability was discovered in `cm.cgi` by modifying the buffer overflow specification of `pathload2`, replacing the source of input from `recv` to `getenv`. This revealed a buffer overflow vulnerability resulting from a string passed through the `QUERY_STRING` HTTP parameter that overflows in `strcpy`.

5) Lighttpd: `Lighttpd` is a popular, high performance webserver that runs on embedded devices and servers alike. We discovered a vulnerability in `lighttpd` resulting from unchecked return values. Although not all instances are exploitable, Saluki reports 75 flaws where `calloc` is unchecked. Our detection and disclosure of these issues prompted the `Lighttpd` maintainers to perform checking of over `calloc` call sites.[8]

Disclosure: We have responsibly disclosed all vulnerabilities. Two vulnerability discoveries remain unresolved. For `easyconf`, the vendors acknowledged the receipt of this issue, but have not responded to further follow-up queries.As mentioned, the author of `pathload2` could not be contacted. For `cm.cgi`, receipt of our reports were not acknowledged after multiple attempts to make contact.

D. Controlled Experiment Case Study: C++ binaries and SQL Injection

C++ compiled binaries present significant challenges to static binary analysis. Vtables and object references add an additional layer of complexity that makes complete CFG recovery and alias analysis difficult. The design choices in §III allow us to analyze security properties in localized areas that works in the presence of partial CFG recovery and without performing a direct alias analysis.

We investigated two additional SQL injection CVEs reported in a media server COTS product. [9] The report names two vulnerable CGI binaries, `watchstatus.cgi` and `audiotrack.cgi`. Saluki detects three cases of SQL injection in the shared library that map to both CVEs with a

[8] Commit 566cf.

[9] https://packetstormsecurity.com/files/133519/Synology-Video-Station-1.5-0757-Command-Injection-SQL-Injection.html

single specification. The specification is listed in Appendix C6. This specification generated false positive. We investigated the false positive and found that it is due to an internal constant string being appended. For true positives, the string being appended is an unbound variable in the function. We demonstrate this example as a validation that (a) can cope with C++ compiled binaries, (b) can be adapted according to check that certain property invariants hold (a SQL string should not be appended using append), and (c) works on a realistic binary intended to evaluate vulnerability detection.

E. Controlled Experiment Case Study: Heartbleed

We demonstrate that Saluki works on large, real-world programs by finding the Heartbleed vulnerability in OpenSSL (378,691 source LOCs). Recall that the root cause of the Heartbleed vulnerability is a missing bounds check on the length argument payload passed to memcpy (line 23 in Listing 5). The attacker-controlled payload can be exploited to leak 64KB of memory to the client.

```
1   // ssl/d1_both.c
2   int dtls1_process_heartbeat(SSL *s) {
3     unsigned char *p = &s->s3->rrec.data[0];
4     unsigned short hbtype;
5     unsigned int payload;
6     ...
7     /* Read type and payload length first */
8     hbtype = *p++;
9     n2s(p, payload);
10  + if (1 + 2 + payload + 16 > s->s3->rrec.length)
11  +  return 0; /* silently discard per RFC 6520 sec.4*/
12    pl = p;
13    ...
14    if (hbtype == TLS1_HB_REQUEST){
15      unsigned char *buffer, *bp;
16      int r;
17      ...
18      buffer = OPENSSL_malloc(1 + 2 + payload + padding);
19      bp = buffer;
20      ...
21      memcpy(bp, pl, payload);
22      ...
23      r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT,
24            buffer, 3 + payload + padding);
25      ...
26    }
27    ...
28  }
```

Listing 5: The Heartbleed Vulnerability.

Statically detecting the Heartbleed vulnerability poses a challenge because it is hard to determine information flow from the original source in the presence of indirect control flow due to function pointers. Source-based approaches infer the n2s macro on line 10 as an input source [40]. The macro is replaced with instructions in the binary code, which mean we can't make use of the inferred source as an API call in a Saluki specification. We note that automatic inference of sources is an interesting research direction that couples well with Saluki. Unfortunately, current approaches have limited applicability for binaries. We leave this topic as a point of future work.

We use a systematic approach that is both checkable by Saluki and overcomes the limitations above. We use the fact that the first basic block contains local registers (including the stack and offsets) that are initially *free* (unbound). The intuition is that these free variables correspond to variables and arguments in source code. We seed taint these free variables in the IR for all functions in OpenSSL using μflux and make use of Saluki's built-in predicate to check whether there is a flow from any tainted free variable to the third argument of memcpy without a check (Specification C9). The check finds exactly the two violations due to Heartbleed: one in function dtls1_process_heartbeat and one in tls1_process_heartbeat. Note that the Heartbleed fix inserts an essential conditional check on payload (lines 11 and 12). We also confirmed that the fix introduces code satisfying the constraint in the Saluki specification, i.e., Saluki does not report an error for the patched binary (as desired).

Limitations: Table I shows low coverage for OpenSSL using μflux. This is due to bounded execution and lack of complete CFG recovery. The implication is that not all of the information flows to memcpy's in OpenSSL are covered. Additionally, the ratio of data dependence facts to instructions is high due to our principled (but generous) taint strategy. We observed 4 false positives due to overtainting, where tainted values flow to memcpy calls that are not reachable. Many of the extracted data dependence facts are unimportant to discovering Heartbleed, but they do not produce overly noisy results with respect to the specification. Our case study validates two important points: Saluki scales to real programs and can express specifications sufficient to find critical vulnerabilities like Heartbleed.

F. Controlled Experiment Discussion

We also analyzed known vulnerabilities in the Linux Kernel (CVE-2013-4513) and Pidgin, a chat client (CVE-2013-6482).

Linux Kernel: Saluki has support to check compiled kernel modules. Compiled kernel modules add complexity to static binary analysis by requiring recovery of relocatable symbols and fixing branch targets. We used Saluki to find a kernel buffer overflow in the ozwpan module due to an absent bounds check on the third argument to copy_from_user. The specification is the same as Heartbleed, but with copy_from_user as the sensitive sink.

Pidgin: The CVE-2013-6482 vulnerability in Pidgin contains a vulnerability where NULL may be passed to atoi from an attacker-controlled API function. In the libmsn.so binary, atoi is compiled to strtol. Specification C7 finds the vulnerability.

C++: Saluki can cope with C++ compiled binaries and can successfully track taint values to detect known vulnerabilities. One challenge with out-of-box C++ support is modeling C++ APIs and compiler ABIs. For example, to find known vulnerabilities in C++, we needed to model (a) that the std::swap function propagates tainted to (b) particular registers not modeled in the default ARM ABI. We note that future improvements to ABI and API promise to benefit C++ support.

Note that Table I shows that non-deterministic (flood) execution mode is successful in finding all of the vulnerabilities, whereas deterministic mode found vulnerabilities in four cases. One threat to validity is that we achieved low coverage for these binaries. However, low coverage may also be due to specifications that only match and execute terms in leaf functions (e.g., ozwpan.ko).

G. Speed and Coverage

We performed several experiments to characterize the speed and coverage of Saluki.

Speed: Table I summarizes the times to analyze vulnerable binaries. Most binaries finish under 100 seconds except for OpenSSL.To determine the speed of μflux more generally, we ran a benchmark over 100 Coreutil binaries. We bounded execution to 10 million IR basic blocks, and initiated execution from each function entry point. The average time of execution per binary is 54 seconds. The time for each binary is shown in Figure 5.

In total, our Coreutils benchmark evaluated roughly 1.5 billion IR instructions in under 82 minutes. This allows us to emulate roughly 320K IR instructions per second with μflux. Though desirable, it is difficult to compare μflux to the MicroX implementation [20], a conceptually similar technique which also enable execution from arbitrary program points. Two main caveats hinder a direct comparison. First, MicroX uses dynamic library instrumentation to execute native x86 instructions, while Saluki implements an IR interpreter (i.e., one x86 instruction may correspond to many IR instructions). Second, MicroX tests on different hardware and targets the x86 `ntdll.dll` binary, while we benchmarked against ARM Coreutils.

Keeping these caveats in mind, we note the MicroX paper achieves roughly 184 instructions per second (taking the number of tests executed times the maximum number of unique instructions as an upper bound), and 127 instructions average across all tests (based on a one minute timeout for each test). Even with conservative approximations of tens of IR instructions per native instruction, our benchmarks are a positive indication that Saluki is fast, with large potential speedup over existing implementations for executing on realistic binaries.

Coverage: Saluki achieves an average statement coverage of 79% over all vulnerable binaries using μflux in non-deterministic mode. In this mode, greater coverage allows μflux to recover more data dependency facts. In our experiments, activating non-deterministic mode (**fl.** flood coverage in Table I) allows Saluki to discover six vulnerabilities (shaded in Table I) that deterministic (**det** coverage) mode cannot.

In our Coreutils benchmark, Saluki achieves 96% statement coverage on the lifted IR averaged over all binaries. Figure 5, in the Appendix,summarizes the benchmark. Binaries which did not achieve 100% coverage occurs when execution is unable to proceed past a loop with the given execution bound (10 loops by default).

VII. Related Work

Taint analysis [33] is a general technique for propagating dataflow information. Shankar et. al. [35] perform a static taint analysis to detect format string vulnerabilities. Our approach for resolving data dependency in this paper is μflux. μflux relates to MicroX [20], a runtime VM for testing purposes. MicroX is a VM for performing dynamic execution from a user-specified function or code location. We differ from MicroX by performing evaluation on the IR of the program, and do not require a native runtime environment to simulate the dynamic behavior of the program. This allows us to seamlessly reuse security property specifications for multiple architectures (ARM, x86, etc.).

Many static analysis techniques have been used for finding security issues in programs. Query language approaches relate closely to the specification aspect of Saluki. Source-based approaches (e.g., by Yamaguchi et al. [39, 40], and tools such as PQL [26, 28, 29] and Pidgin [24] use taint-style patterns to find vulnerabilities. In general, binaries present unique challenges where we cannot exploit knowledge of source-level constructs (e.g., macros and objects). We adapt to the challenge by reasoning exclusively over the semantics of binary code to generate data dependence facts, combined with a formal model to find vulnerabilities. Automatic inference (as in [40] is a promising avenue that could be used in conjunction with Saluki to automatically inferring vulnerability specifications). Metal [18, 22] uses rule templates for finding defects in source code; in this context, Saluki specifications also performs user-supplied assertion checks, but for binary code. Cova et. al. [15] present a taint-based symbolic execution technique to find taint-style vulnerabilities for binaries. However, this work lacks a formal approach to modeling vulnerabilities and cannot describe vulnerabilities using reusable specifications.

Model checking tools and techniques [14, 19, 30] such as MOPS [12, 34] and SLAM [5, 6] verify temporal safety properties of programs. Saluki can be seen as taking a model checking approach which elides the need for traditional model checking connectives by introducing the single data dependence relation. Dynamic techniques such as concolic execution [21] and fuzzing [31] are effective at finding vulnerabilities, but are restricted to analyzing the paths observed during runtime execution. Although we presented a static analysis, we envision dynamic techniques as complementary to our approach. For one, we can refine the data dependency model in Saluki based on execution traces and dynamic coverage.

VIII. Conclusion

We introduced Saluki, a new tool for checking taint-style (data dependence) security properties in binary code. Saluki comprises (a) a novel logic system and property language to express security properties and (b) a new technique called μflux for extracting path- and context-sensitive data dependencies. A user-supplied specification drives a decidable analysis over the program model that outputs a proof of property violations. We demonstrated our ability to describe a wide variety of vulnerabilities with Saluki specifications. Our specifications describe a number of CWEs demonstrating the practical value of our approach. We used Saluki to find known vulnerabilities in the Linux Kernel, OpenSSL, and C++ compiled binaries, as well as five 0-day vulnerabilities in COTS binaries. Our results suggest that Saluki can be applied effectively toward vulnerability discovery. We used Saluki to model the behavior of binary programs, but recognize that our formal system can be applied to source-based techniques as a point of future work. A further promising avenue for future work is to evaluate the benefits of different backend analyses for generating data dependence facts under Saluki.

IX. Acknowledgments

This work is partially supported under DARPA grant number N66001-13-2-4040 and the Korean IoTcube project. All statements are those of the authors, and do not necessarily reflect the views of the funding agency.

References

[1] "Binary Analysis Platform," https://github.com/BinaryAnalysisPlatform/bap, 2016, online; accessed 7 April 2016.

[2] "Coreutils - GNU core utilities," http://www.gnu.org/software/coreutils/coreutils.html, 2016, online; accessed 16 February 2016.

[3] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing Symbolic Execution with {Veritesting}," in *International Conference on Software Engineering*, New York, New York, USA, 2014, pp. 1083–1094.

[4] G. Balakrishnan and T. Reps, "WYSINWYX: What You See Is Not What You Execute," *ACM Transactions on Programming Languages and Systems*, no. 6, pp. 1–84, 2010.

[5] T. Ball, V. Levin, and S. K. Rajamani, "A Decade of Software Model Checking with SLAM," *Communications of the ACM*, vol. 54, no. 7, p. 68, 2011.

[6] T. Ball and S. K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," in *Symposium on Principles of Programming Languages*, 2002, pp. 1–3.

[7] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "ByteWeight: Learning to Recognize Functions in Binary Code," in *USENIX Security Symposium*, 2014, pp. 845–860.

[8] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A taint based approach for smart fuzzing," *IEEE International Conference on Software Testing, Verification and Validation*, pp. 818–825, 2012.

[9] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," Microsoft MSR-TR-2012-55, Tech. Rep., 2012.

[10] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "{BAP}: A Binary Analysis Platform," in *International Conference on Computer Aided Verification*, 2011, pp. 463–469.

[11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[12] H. Chen and D. Wagner, "MOPS: An Infrastructure for Examining Security Properties of Software," in *ACM Conference on Computer and Communications Security*, 2002, pp. 235–244.

[13] V. Chipounov, V. Kuznetsov, and G. Candea, "{S2E}: A Platform for In-Vivo Multi-Path Analysis of Software Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.

[14] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: algorithmic verification and debugging," *Communications of the ACM*, vol. 52, no. 11, pp. 74–84, 2009.

[15] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static Detection of Vulnerabilities in x86 Executables," *IEEE Annual Computer Security Applications Conference*, pp. 269–278, 2006.

[16] J. Dahse, G. Horst, and T. Holz, "Simulation of Built-in PHP Features for Precise Static Code Analysis," in *Proceedings of the Network and Distributed System Security Symposium*, 2014, pp. 23–26.

[17] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[18] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules Using System-Specific Programmer-Written Compiler Extensions," in *USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[19] P. Godefroid and N. Klarlund, "Software Model Checking: Searching for computations in the abstract or the concrete," *Integrated Formal Methods*, 2005.

[20] P. Godefroid, "Micro execution," *International Conference on Software Engineering*, pp. 539–549, 2014.

[21] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[22] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," *ACM SIGPLAN Notices*, vol. 37, no. 5, p. 69, 2002.

[23] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *USENIX Security Symposium*, 2013.

[24] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," *Programming Language Design and Implementation*, pp. 291–302, 2015.

[25] N. Jovanovic, C. Kruegel, and E. Kirda, "Static analysis for detecting taint-style vulnerabilities in web applications," *Journal of Computer Security*, vol. 18, no. 5, pp. 861–907, 2010.

[26] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive Program Analysis as Database Queries," *ACM Symposium on Principles of Database Systems*, pp. 1–12, 2005.

[27] W. Landi, "Undecidability of Static Analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, dec 1992.

[28] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *USENIX Security Symposium*, 2005.

[29] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a Program Query Language," in *ACM SIGPLAN Notices*, vol. 40, no. 10, 2005, p. 365.

[30] M. Musuvathi, D. Park, and A. Chou, "CMC: A pragmatic approach to model checking real code," 2002.

[31] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing Seed Selection for Fuzzing," in *USENIX Security Symposium*, 2014, pp. 861–875.

[32] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld, "Explicit secrecy: A policy for taint tracking," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 15–30.

[33] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[34] B. Schwarz, D. Wagner, J. Lin, G. Morrison, and J. West, "Model Checking An Entire Linux Distribution for Security Violations," *Annual Computer Security Applications Conference*, pp. 13–22, 2005.

[35] U. Shankar, Talwar Kunal, J. S. Foster, and D. Wagner, “Detecting Format String Vulnerabilities with Type Qualifiers.” in *USENIX Security Symposium*, 2001.

[36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “Sok: (state of) the art of war: Offensive techniques in binary analysis,” in *IEEE Symposium on Security and Privacy*, 2015.

[37] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller : Augmenting Fuzzing Through Selective Symbolic Execution,” pp. 21–24, 2016.

[38] J. Wilhelm and T.-c. Chiueh, “A Forced Sampled Execution Approach to Kernel Rootkit Identification,” in *International Symposium on Recent Advances in Intrusion Detection*, 2007, pp. 219–235.

[39] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and Discovering Vulnerabilities with Code Property Graphs,” in *IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[40] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, “Automatic inference of search patterns for taint-style vulnerabilities,” in *IEEE Symposium on Security and Privacy*, 2015, pp. 797–812.

Appendix

A. *Benchmark*

B. *Taint Propagation Rules*

$$\boxed{\begin{array}{ll} a \rightsquigarrow v & \text{load } v \text{ from } a \\ *a := v & \text{store } v \text{ at } a \\ v \mapsto t & v \text{ is tainted by } t \end{array}}$$

$$\frac{a \rightsquigarrow v \qquad a \mapsto t}{v \mapsto t}\ \text{LOAD} \qquad\qquad \frac{*a := v \qquad v \mapsto t}{a \mapsto t}\ \text{STORE}$$

$$\frac{\Diamond_u v_1 \rightsquigarrow v_2 \qquad v_1 \mapsto t}{v_2 \mapsto t}\ \text{UNOP} \qquad\qquad \frac{v_1 \Diamond_b v_2 \rightsquigarrow v_3 \qquad v1 \mapsto t}{v_3 \mapsto t}\ \text{LHS}$$

$$\frac{v_1 \Diamond_b v_2 \rightsquigarrow v_3 \qquad v2 \mapsto t}{v_3 \mapsto t}\ \text{RHS}$$

Fig. 6: A Taint Semantics for Concrete Evaluation

C. *Vulnerability Specifications*

A summary of the vulnerability specifications we used are provided here, with associated CWE ID and description.

1) Specification 1: CWE-120: “Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)” with `recv`.

```
prop recv_to_strcpy ::=
  recv (_,*p,_,_), strcpy (_,*q) |- never
  s.t. {q / p}
```

Listing 6: `pathload2` specification

2) Specification 2: CWE-120: “Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)” with `getenv`.

```
prop recv_to_strcpy ::=
  p = getenv (_), strcpy (_,*q) |- never
  s.t. {q / p}
```

Listing 7: `cm.cgi` specification

3) Specification 3: CWE-78: “Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’).”

```
prop recv_to_system ::=
  sprintf (*p,_,_), system (*q) |- never
  s.t. {q / p}
```

Listing 8: admin.cgi specification

4) Specification 4: CWE-252: “Unchecked Return Value.”

```
prop calloc_maybe_checked ::=
  p := calloc(_) |- when c jmp _
  s.t. {c / p}
```

Listing 9: `Lighttpd` specification

5) Specification 5: CWE-337: “Predictable Seed in PRNG.”

```
prop srand_seeded_with_time ::=
  p := time(_), srand(q) |- never
  s.t. {q / p}
```

Listing 10: `admin.cgi` specification

6) Specification 6: CWE-89:

```
prop no_sql_appends_before_add_cond ::=
  _ZNSs6appendEPKcj(p,_),
  j__ZN15LibVideoStation7VideoDB11AddCondtionERKSs
  |- never
  s.t. {q / p}
```

Listing 11: No SQL appends before SQL create

Note that Listing 11 uses the mangled C++ names of methods to verify the property with Saluki.[10]

7) Specification 7: CWE-676:

```
prop no_check_on_strtol_from_source ::=
  p = xmlnode_get_data (_),
  strtol(q,_,_)
  |- when c jmp _
  s.t. {c / p, q /p}
```

Listing 12: No check when passing input to `strtol`

8) Specification 8: CWE-120:

```
prop taint_to_memcpy_with_no_check ::=
  p = _;
  memcpy(_,_,q)
  |- when c jmp _
  s.t. {q / p, c / p, for all p that is tainted}
```

Listing 13: No bounds check on third argument of `memcpy`

[10] Mangled C++ names may be demangled with the `c++filt` utility. E.g., `_ZNSs6appendEPKcj` produces `std::basic_string<char, std::char_traits<char>, std::allocator<char»::append(char const*, unsigned int)`

$$\frac{(t \models p\,\mathbf{s.t.}\,c) \to (t,t' \models p,p'\,\mathbf{s.t.}\,c)}{t,t' \models \mathbf{prop}\,id \triangleq p \vdash p'\,\mathbf{s.t.}\,c}\;(\texttt{PROP}) \qquad \frac{t\sharp\langle p\rangle \rightsquigarrow G \qquad G\models c}{t\models p\,\mathbf{s.t.}\,c}\;(\texttt{PATTS}) \qquad \frac{t\sharp\langle p\rangle \rightsquigarrow G \qquad t'\sharp\langle p\rangle\rightsquigarrow G'}{(t,t')\sharp\langle p\rangle \rightsquigarrow G,G'}\;(\texttt{TERMS}) \qquad \frac{t\sharp\langle p\rangle\rightsquigarrow G}{t\sharp\langle p,p'\rangle \rightsquigarrow G}\;(\texttt{LHS-P})$$

$$\frac{t\sharp\langle p'\rangle\rightsquigarrow G}{t\sharp\langle p,p'\rangle\rightsquigarrow G}\;(\texttt{RHS-P}) \qquad \frac{}{(l,\_,\_,\_,S,S')\sharp\langle v := v'\rangle \rightsquigarrow (v \overset{l}{=} S, v' \overset{l}{=} S')}\;(\texttt{DEF}) \qquad \frac{S = (\_,\_,\_,\dots,\_)}{(l,S,S',\_,\_)\sharp\langle \mathbf{when}\, v\, \mathbf{jmp}\, v'\rangle \rightsquigarrow (v\overset{l}{=}S', v'\overset{l}{=}S)}\;(\texttt{JMP})$$

$$\frac{R_1 = id \vee \dots \vee R_n = id}{(l,(R_1,\dots,R_n),\_,\_,\_)\sharp\langle v_0 := id(v_1,\dots,v_m)\rangle \rightsquigarrow v \overset{l}{=} \arg(id,0),\dots,v_m \overset{l}{=} \arg(id,m)}\;(\texttt{SUB})$$

$$\frac{\mathcal{D}(l',R',l,R) \qquad G(v)\overset{l}{\mapsto} R \qquad G(v')\overset{l'}{\mapsto}R'}{G\models v'/v}\;(\texttt{DEP}) \qquad \frac{G(v)\overset{l}{\mapsto}R \qquad P(\mathcal{P},l,R)}{G\models \mathcal{P}(v)}\;(\texttt{PRE}) \qquad \frac{G(v)\overset{l}{\mapsto}\top}{G\models v'/v}\;(\texttt{LHS}) \qquad \frac{G(v')\overset{l'}{\mapsto}\top}{G\models v'/v}\;(\texttt{RHS}) \qquad \frac{G(v)\overset{l}{\mapsto}\top}{G\models\mathcal{P}(v)}\;(\texttt{TOP})$$

$$\frac{}{()(v)\overset{l}{\mapsto}\top}\;(\texttt{NIL}) \qquad \frac{(R=R_1)\vee\dots\vee(R=R_m)}{(v\overset{l}{=}(R_1,\dots,R_m),G)(v)\overset{l}{\mapsto}R}\;(\texttt{HD}) \qquad \frac{G(v)\overset{l}{\mapsto}R \qquad v\neq v'}{(v\overset{l}{=}R',G)(v)\overset{l}{\mapsto}R}\;(\texttt{TL})$$

Fig. 4: The Saluki Inference Rules

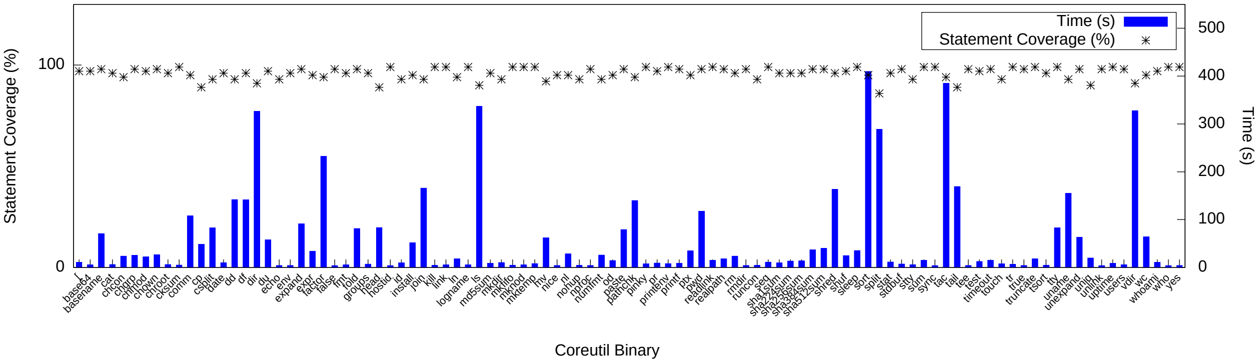

Fig. 5: GNU Coreutils Benchmark

9) Specification 9: CWE-120:

```
prop taint_to_copy_from_user_with_no_check ::=
  p = _;
  copy_from_user(_,_,q)
  |- when c jmp _
  s.t. {q / p, c / p, for all p that is tainted}
```

Listing 14: No bounds check on third argument of `memcpy`

D. Proof System

We prove properties specified in the Saluki language using a set of inference rules (see Figure 4). Given a program $\mathbb{P} = (\mathcal{T}, \mathcal{D}, \mathcal{P})$ and a property P, our goal is to prove that the property holds for all possible combinations of terms,

$$\forall t_1 \dots \forall t_m, \mathcal{T}t_1 \wedge \dots \wedge \mathcal{T}t_m \to t_1, \dots, t_m \models P.$$

For each combination of terms we build a derivation tree. If for some sequence of terms it is not possible to build the tree, then we use this sequence as a constructive proof that the property does not hold. To prove a property P named id for a sequence of terms t, t' we must build the derivation from $t, t' \models \mathbf{prop}\, id \triangleq p \vdash p'\,\mathbf{s.t.}\,c$. We say that t, t' proves property id if it is possible to build the tree.

Property rule: The `(prop)` rule proves either that (1) $t \models p\,\mathbf{s.t.}\,c$ doesn't hold or (2) $t, t' \models p, p'\,\mathbf{s.t.}\,c$ holds. Both are proved with the `(patts)` rule. The `(patts)` rule obligates the solver to find a valuation G from all possible valuations of t, denoted $t\sharp\langle p\rangle \rightsquigarrow G$, satisfying the constraint c.

Term rules: Rule `(terms)` corresponds to the `(comma-c)` axiom, and obligates that valuations from both terms should be taken into account. However, the rule gives the solver freedom to choose the order of matching. A pair of rules `(lhs-p)` and `(rhs-p)` directly corresponds to the `(comma-p)` axiom.

Rules `(def)`, `(jmp)`, and `(sub)` define how terms valuate the corresponding patterns. They are a straightforward implementation of the corresponding axioms. There is no rule for the **never** pattern, so it is impossible to build a derivation for it.

Constraint rules: Rules `(dep)` and `(pre)` obligate the solver to find bindings in valuation a G for which $\mathcal{D}$ and $\mathcal{P}$ propositions hold, respectively.

The proposition $G(v) \overset{l}{\mapsto} R$ is defined by rules `(nil)`, `(hd)`, and `(tl)` which inducts over a list of bindings. These rules are used to prove that constraints $\mathcal{D}$ and $\mathcal{P}$ are satisfied (cf., rules `(dep)`, `(pre)`, `(rhs)`,`(lhs)`,`(top)`).

Correctness: The Completeness Theorem states that our inference rules are consistent with a set of axioms that define the language semantics. Our formalism lays the groundwork for proving correctness properties. We provide an outline of proofs for our formalism, but do not pursue them fully. Recall that our objective is foremost to implement principled reasoning and checking of data dependence facts.

Soundness Part: Soundness states that our inference rules preserves the truth, i.e., that our system produces only tautologies.

Proof. The proof is an induction over the length of a derivation tree. In the previous section we showed the relation of the semantics to axioms. All of the base cases in our semantics are a straightforward implementation of corresponding axioms.

Completeness Part: Completeness states that our inference rules are consistent with a set of axioms that define the language semantics. To prove completeness, we want to show that our system produces not only tautologies, but all of the tautologies.

Proof Sketch. Our formalism lays the groundwork to show that it is complete. However, proving completeness is a laborious exercise that is typically done using a automated theorem prover such as Coq. The proof may be sketched out as follows: First assume that there exists a derivation of rules for which a corresponding proposition does not hold. Then show that by visiting each possible branch of the derivation tree we reach a contradiction. A contradiction implies that our rules are consistent. This step is very tedious as it must be done for each possible derivation. We leave this as an exercise for future work.

E. Coreutils Benchmark

Refer to Figure 5.