Formal Semantics for the DTrace Tracing System
Student: Domagoj Stolfa
Advisors: Robert N. M. Watson, Alan Mycroft
University of Cambridge

Background and Motivation

Tracing systems, tools originally designed to debug and profile the performance of large systems with minimal disruption, are now used increasingly for behaviour monitoring [1, 5] and security auditing [2, 6]. Because of security-critical use and complex semantics, any formal description of such a system is required. In this work we target the most widely deployed tracing system, DTrace, and formalise its semantics in HOL4 [3].

DTrace accepts scripts written in D (below); these express probes and are compiled to bytecode (actions and DTrace Intermediate Format (DIF)) in consumers. The following script tracks write operations to a Unix file descriptor (files, standard output, socket, etc.):

```d
syscalls::write:entry
{ self->rd = arg0; sttaa x, rd (DIF code) }
```

DTrace accepts scripts written in D (below); these express probes and are compiled to bytecode (actions and DTrace Intermediate Format (DIF)) in consumers. The following script tracks write operations to a Unix file descriptor (files, standard output, socket, etc.):

```d
syscalls::write:entry
{ self->rd = arg0; sttaa x, rd (DIF code) }
```

DTrace probes fire concurrently (even self-concurrently) and can non-deterministically discard data and stop executing any further code as a result of a failure (we denote this as FAIL in our presentation). We model them as S-Probes (concurrent processes) and present their run-time semantics as a labelled transition system of form $P \xrightarrow{c} P'$. Dynamic D variables (compiled to DIF variables in probes) have complex semantics (see forthcoming paper for full definition) and can be either global or thread-local. Reading an unmapped variable returns the value $\bot$, while allocation is performed upon writing a value to a variable the first time. Allocation may fail if name resolution (e.g. an associative array) has been loaded into the probe.

```
syscalls::write:entry
{ self->rd = arg0; sttaa x, rd (DIF code) }
```

The Formal Model

<table>
<thead>
<tr>
<th>$P, Q, R :=$ Semantic Probes (S-Probes)</th>
<th>$a :=$ Transition labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>null probe</td>
</tr>
<tr>
<td>$P \parallel Q$</td>
<td>parallel composition</td>
</tr>
<tr>
<td>$P \parallel Q$</td>
<td>sequential composition</td>
</tr>
<tr>
<td>$P \rightarrow t$</td>
<td>replication</td>
</tr>
</tbody>
</table>

DTrace probes fire concurrently (even self-concurrently) and can non-deterministically discard data and stop executing any further code as a result of a failure (we denote this as FAIL in our presentation). We model them as S-Probes (concurrent processes) and present their run-time semantics as a labelled transition system of form $P \xrightarrow{c} P'$. Dynamic D variables (compiled to DIF variables in probes) have complex semantics (see forthcoming paper for full definition) and can be either global or thread-local. Reading an unmapped variable returns the value $\bot$, while allocation is performed upon writing a value to a variable the first time. Allocation may fail if name resolution (e.g. an associative array) has been loaded into the probe.

```
syscalls::write:entry
{ self->rd = arg0; sttaa x, rd (DIF code) }
```

Insights — DTrace as a Protocol

Let $P$ be an arbitrary S-Probe. We define a happens-before relation on a static channel $s$ and $P$ to be:

$$P \parallel s \vdash P'$$

```
syscalls::write:entry
{ self->rd = arg0; sttaa x, rd (DIF code) }
```

We use that to define data races with respect to $s$ and $P$ as $P \parallel s \vdash P' \land P' \parallel s \vdash P$. We then present the Formal Model as an Implementation

While the formal model itself gives a better understanding of DTrace, we have started leveraging the existing HOL4 implementation of the formal model in order to generate executable Standard ML. We plan to use that in order to implement symbolic execution and use that as a test oracle for DTrace implementations. Moreover, we hope to implement NuD, a safer exploratory language as an alternative to D which is more amenable to automated reasoning. We have started formulating a type system inspired by multi-party session types for which NuD will serve as an implementation.

```
syscalls::write:entry
{ self->rd = arg0; sttaa x, rd (DIF code) }
```

Acknowledgements and References

This work is part of the CADETS Project sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-15-C-7558. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.