TESLA: Temporally Enhanced System Logic Assertions

Jonathan Anderson
Robert N. M. Watson David Chisnall
Khilan Gudka Ilias Marinos
University of Cambridge
firstname.lastname@cl.cam.ac.uk

Brooks Davis
SRI International
brooks@csl.sri.com

Abstract

Large, complex, rapidly evolving pieces of software such as operating systems are notoriously difficult to prove correct. Developers instead describe expected behaviour through assertions and check actual behaviour through testing. However, many dynamic safety properties cannot be validated this way as they are temporal: they depend on events in the past or future and are not easily expressed in assertions.

TESLA is a description, analysis, and validation tool that allows systems programmers to describe expected temporal behaviour in low-level languages such as C. *Temporal assertions* can span the interfaces between libraries and even languages. TESLA exposes run-time behaviour using program instrumentation, illuminating coverage of complex state machines and detecting violations of specifications.

We apply TESLA to complex software, including an OpenSSL security API, the FreeBSD Mandatory Access Control framework, and GNUstep's rendering engine. With performance allowing "always-on" availability, we demonstrate that existing systems can benefit from richer dynamic analysis without being re-written for amenability to a complete formal analysis.

1. Introduction

It is notoriously difficult to prove large, complex, rapidly evolving pieces of software such as operating systems correct. Low-level software often uses techniques such as function pointers and non-polymorphic casting that increase the difficulty of static analysis. Even simple temporal properties (e.g., "a permission check was done earlier in this system call") are obscured by indirection and dynamic modules.

EuroSys 2014, April 13–16 2014, Amsterdam, Netherlands.

ACM 978-1-4503-2704-6/14/04…$15.00.
http://dx.doi.org/10.1145/2592798.2592801

```
void foo(struct object *o, int op) {
  TESLA_WITHIN(enclosing_fn, previously(
    security_check(ANY(ptr), o, op) == 0));
```

Figure 1: A TESLA assertion: within the execution of `enclosing_fn`, a previous call to `security_check` with arguments (unspecified pointer, `o`, `op`) should have returned `0`.

Most existing systems were not written with formal analysis in mind and retrofitting this style is a daunting task. Notwithstanding the partial verification of seL4 [19], formal analysis will not be part of commodity systems development workflows for the foreseeable future. Static analysis of whole programs can prove properties about all possible executions, but we are sometimes less interested in learning if events are *possible* than how often they occur in *practice*.

Without formal analysis techniques, developers use a variety of techniques to improve the reliability of their code. One tool is the *assertion*: a statement made by the developer, in the same language as their code, expressing assumptions about system state. Assertions can document pre- and post-conditions and invariants, to be checked at run time. They describe only instantaneous properties: e.g., about a current value, but not about events that should have occurred in the past (e.g., "an access-control check should have been performed") or the future (e.g., "this memory will be freed before that lock is released"). Some systems incorporate ad hoc mechanisms to test temporal properties (e.g., FreeBSD's WITNESS lock-order verifier [1]). Hand-crafted checkers are valuable—FreeBSD rarely experiences deadlocks—but are costly to implement and easy to get wrong.

TESLA is a generic tool that lets programmers specify temporal properties they expect their programs to have, leaving run-time checks to be generated mechanically. Using *temporal assertions* such as figure 1, TESLA helps developers of complex systems to describe intended temporal behaviour, understand actual run-time behaviour and detect mismatches between the two.

Using TESLA, we were able to observe implementation errors in OpenSSL, FreeBSD and GNUstep, discovering new security bugs in FreeBSD and its test suite.

```
  X509_STORE_CTX_set_cert(&xsc,x);
- if (!reqfile && !X509_verify_cert(&xsc))
+ if (!reqfile && X509_verify_cert(&xsc) <= 0)
    goto end;
```

Figure 2: API misuse example from CVE-2008-5077: OpenSSL's `x509` code incorrectly checks the return value of `X509_verify_cert`, conflating `-1` (error) with `1` (success).

2. Use cases

We illustrate TESLA's motivation, use, and performance via three complex systems: OpenSSL clients' use of an error-prone API, the FreeBSD Mandatory Access Control implementation, and GNUstep's stateful APIs.

2.1 Correct usage of a security API

In 2008, Google's security team found that core OpenSSL applications were vulnerable to abuse due to erroneous use of OpenSSL APIs [32]. Many OpenSSL APIs have tri-state return codes: 1 indicates success, 0 indicates failures, and -1 indicates an exceptional error. Several OpenSSL-based applications checked return values with expressions that conflated success and exceptional errors (figure 2).

In this example, if `X509_verify_cert` returns `-1` on an ASN.1 type error, the `openssl x509` command behaves as if the certificate has been verified when it has not. This trivial example can be detected by tools such as Coccinelle [23] or even UNIX `grep`, but only when a static function is directly called by client code. Some erroneous calls were embedded in a wrapper library. Client code such as FreeBSD's `libfetch` calls OpenSSL's `libssl`, which incorrectly called `libcrypto` functions. In this case, a `libfetch` developer must not only vet `libfetch` but also all libraries it depends on. When these libraries employ macros, or call verification code via function pointers that may be overridden at run time, text-based tools prove inadequate.

On learning of such a vulnerability, it is desirable for downstream developers to be able to write a single description of the expected behaviour ("previously, the certificate was *correctly* verified") and leave checking work—including for library code—to the compiler without needing to examine library code or alter existing abstractions and APIs. TESLA's dynamic validation allows this workflow, even when function-pointer indirection occurs, but at the cost of instrumentation and a runtime support library.

2.2 Correct implementation of a security framework

Many critical operating-system security properties are temporal safety properties, e.g., the safe reuse of memory, access-control checks being performed before object use, and the eventual auditing of security events. To illustrate the use of TESLA for this purpose, we applied it to the widely-used open-source FreeBSD operating system [28].

FreeBSD implements several types of access control, including Mandatory Access Control (MAC). The FreeBSD MAC Framework separates mechanism—hooks throughout the kernel declaring when subjects operate on objects—from the specific policies that govern a system. The framework has been used to implement historic security policies such as Bell-LaPadula [3] and Biba [5] as well as modern security policies such as those in iOS and Mac OS X [35].

Due to the kernel's modularity and object-oriented approach to C, access-control checks can be distant from the operations they govern. For example, figure 3 illustrates how `fo_poll`, used by the `poll` and `select` system calls, reaches the protocol-specific implementation, `sopoll_generic`. Given the complex code path from one to the other, passing through multiple function pointers implementing object orientation across open-file types, it would be desirable to statically prove that all possible call paths to `sopoll_generic` within a system call first invoke the relevant MAC function, `mac_socket_check_poll`, to authorise access. However, the kernel's abstractions and dynamic modules make static checking extremely difficult.

TESLA uses a dynamic approach to detect whether such checks are performed correctly. We found multiple bugs: not just missing checks, but also an instance of the correct check made with the wrong arguments. We also detected significant functional omissions in test suites designed to check the correctness of kernel access control.

2.3 Exploration of stateful APIs

Understanding the behaviour of complex programs with hidden state and dynamic flow control is difficult. This applies when the flow control is a property of the language, as in Objective-C, and in dispatch mechanisms built on top of the core language, such as vtable-based C object models. Poor understanding of how concurrent components interact can lead to sub-optimal code, and worse, difficult-to-find bugs.

An example is a subtle GNUstep cursor bug that we identified with TESLA after debugging efforts with conventional tools had failed. This is not a security bug, but it is emblematic of the subtle concurrency errors that often emerge from the complex interactions of loosely-coupled systems and are difficult or impossible to debug with existing tools. Such errors are often the cause of serious security issues.

GNUstep is an open-source implementation of the Objective-C frameworks published in the OpenStep specification and now branded as Cocoa by Apple. Its graphics library maintains a stack of cursor images that clients push to and pop from. In June 2013, a GNUstep user reported that a wrong cursor image occasionally displayed. Despite a detailed report, extensive discussion, and powerful debugging tools, the GNUstep developers were unable to identify the problem's source. Tools such as `valgrind` [31] and emulators were too slow to reproduce the problem in an interactive UI, while debuggers such as `gdb` perturbed the timing needed to trigger this concurrency bug. With TESLA, however, we

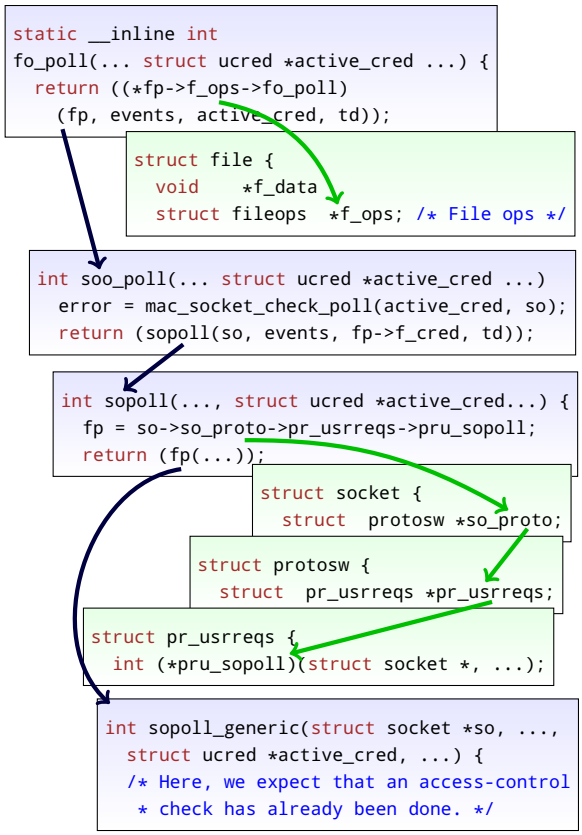


Figure 3: Abstraction layers separate a check from the code it governs with indirection (green) and function calls (blue).

introduced low-cost instrumentation that allowed us to identify the bug, as described in section 3.5.3.

Complex control flow—and a lack of understanding of it—is especially relevant for stateful APIs: for example, the PostScript state machine used in many 2D graphics systems. To avoid hundreds of arguments to function calls, various attributes (stroke colour, transform matrix, and so on) are set independently. Subsequent commands use these properties, so the behaviour of a single draw-line method depends on many previous calls. We investigated a second GNUstep bug related to a new back-end library. The description was vague: things are drawn on the screen incorrectly. Understanding failure cases required introspection on complex temporal behaviour. We could have manually modified roughly 110 methods, with the complexity and error this implies, but instead used TESLA to understand the runtime behaviour in a useful way. These results are described in section 3.5.3.

3. Describing safety properties

TESLA allows programmers to describe safety properties using temporal assertions in their source code. These assertions are inspired by Linear Temporal Logic (LTL) and have a natural expression as finite-state automata [34] that can be mechanically woven into a program (section 4.2).

```
int sopoll_generic(struct socket *so, ...,
  struct ucred *active_cred, ...) {
  TESLA_SYSCALL_PREVIOUSLY(
    mac_socket_check_poll(active_cred, so) == 0);
```

Figure 4: A TESLA assertion that expresses the same safety property as the comment in figure 3.

3.1 Temporal assertions

Temporal assertions augment standard assertions with keywords such as `previously` and `eventually` that allow programmers to specify temporal events relative to the moment the assertion site is reached. A simple example is shown in figure 4 asserting a prior `mac_socket_check_poll` call with the same arguments returned `0`.

TESLA assertions, whose grammar is shown in figure 5, are inspired by LTL, but they are not formally as expressive. For example, TESLA cannot express concepts such as "globally" or "until" that refer to all events over a time period: instead it uses concepts such as `eventually` that can be evaluated with just a few instrumentation points. Even then, expressions such as `previously` are implemented with macros that expand to use low-level reserved symbols such as `__tesla_sequence`. The high-level macros exist purely for programmer convenience: should name collisions arise, alternative macros can be defined or programmers can directly target the low-level names.

Temporal assertions do not run in the same way as instantaneous assertions. Rather, the former expose descriptions of expected behaviour to the TESLA analyser, which emits automata that drive program instrumentation. These descriptions include a context (section 3.2), bounds (section 3.3) and an expression (section 3.4).

3.2 Automata contexts

Temporal assertions allow programmers to describe allowed orderings of program events using automata, as exemplified in figure 9. Automata consume ordered sequences of events, but complex, multi-threaded systems can have arbitrary event interleavings. To adapt real system behaviour to the automaton model, the programmer must specify how program events should be mapped into serial strings of symbols. If the assertion is in an implicitly serialised *context* such as a thread, TESLA can omit internal synchronisation.

TESLA supports two contexts: thread-local and global. In the thread-local context event serialisation is implicit. When describing behaviours that spanning threads, the global context provides explicit synchronisation. This yields a serialisation of events that is non-deterministic but commits to an event order that corresponds to actual program behaviours: an event such as function call entry or exit cannot complete until its instrumentation hook has finished running.

```
assert      :=  'TESLA_GLOBAL(' A ')'
            |   'TESLA_PERTHREAD(' A ')'
            |   'TESLA_ASSERT(' context ',' A ')'
            |   'TESLA_WITHIN(' fnName ',' A ')'
A           :=  start ',' end ',' expr
start       :=  staticExpr
end         :=  staticExpr
expr        :=  modifier '(' expr ')' | boolExpr
            |   sequence | function | fieldAssign
            |   'TESLA_ASSERTION_SITE'
modifier    :=  'optional'
            |   'callee' | 'caller'
            |   'strict' | 'conditional'
boolExpr    :=  expr (op expr)+
op          :=  '||' | '^'
sequence    :=  'TSEQUENCE(' expr (',' expr)* ')'
            |   'previously(' expr (',' expr)* ')'
            |   'eventually(' expr (',' expr)* ')'
function    :=  'call(' fnExpr ')'
            |   'returnfrom(' fnExpr ')'
            |   fnExpr '==' val
fnExpr      :=  fnName '(' args ')'
staticExpr  :=  'call(' fnName ')'
            |   'returnfrom(' fnName ')'
args        :=  val (',' val)*
val         :=  'flags(' C flags ')'
            |   'bitmask(' C flags ')'
            |   C value | 'any(' C type ')'
```

Figure 5: High-level TESLA assertion grammar. This grammar uses the high-level TESLA macros for readability; these macros translate to reserved-namespace C symbols.

3.3 Temporal bounds

In a finite system with many in-flight automata, we cannot allow unbounded `previously` and `eventually` assertions. The programmer must specify bounds at which automata can be initialised and finaliased, e.g., "within the current system call" and "within the current page fault handler".

The bounding requirement allows TESLA to control its memory footprint (described in section 4.4.1). It is possible to use very loose bounds, e.g., `TESLA_WITHIN(main)`, but programmers would be well-advised to only write such assertions when they have a clear understanding of the total number of automata that could be generated.

We have not found the bounding requirement to be onerous; intuitively, the statement "this event must eventually be audited" is only useful when "eventually" is understood to be a specific point in time, well before the system's next reboot, when the truth of the proposition can be evaluated.

3.4 TESLA expressions

Using TESLA, programmers describe the expected system behaviour in terms of temporal relationships among observable events. These descriptions use three types of expressions: concrete events, abstract operations, and modifiers that guide the interpretation of sub-expressions.

3.4.1 Events

TESLA events are concrete program events observable from instrumentation, including function call and return, structure field assignment, and reaching an assertion site.

Function call/return Programmers can specify that they expect a C function to be called (or an Objective-C message to be sent) with particular parameters and/or return a particular value. The specifications in figure 1 says that, within the scope of the function `enclosing_function`, we expect the `security_check` function to have been previously called with parameters `o` and `op` from the scope of the function `foo`. The first parameter `ANY(ptr)` is a wildcard expression: any value passed as the first argument to `security_check` will match. Argument patterns can also be specified as minimal or maximal bitfield, or indirectly using the C address-of operator. This is particularly useful for APIs passing values out by pointer, using return values for error codes.

Function call and return events can be specified with the equality pattern above or with more explicit `call(fn_name(optional_args))` and `returnfrom(...)` patterns.

Field assignment The second concrete event type is assignment to a structure field. Programmers can describe simple assignment (e.g., `s.foo = NEXT_STATE`) or compound assignment (e.g., `s.foo += 1` or `s.foo++`).

Assertion site The final concrete TESLA event type occurs when program execution reaches an assertion site. This event can be described explicitly with `TESLA_ASSERTION_SITE` or implicitly with `previously(x)` or `eventually(x)`, which expand to `[x, TESLA_ASSERTION_SITE]` in the `previously` case and `[TESLA_ASSERTION_SITE, x]` for `eventually`.

3.4.2 Operators

TESLA expressions can also include sequences and boolean operators that describe relationships among events. A sequence of TESLA expressions can be specified with the `TSEQUENCE` operator or with `previously(x)` and `eventually(x)` macros, which expand as described in "Assertion site" above. Each sub-expression can itself be a concrete event or a complex expression.

TESLA also supports the boolean OR operator. Finite-state automata model regular languages with sequences, repetition, and the exclusive-or operator. In the assertion `previously(check(x)||check(y))`, it is not an error for *both* checks to be performed. Rather, the logical OR $\vee$ stipulates that at least one occurred. We implement $\vee$ by constructing an automaton that tracks the state of both original automata independently in a cross-product–like operation:

$$states(a \vee b) = \{a_i b_j \mid a_i \in a \; and \; b_j \in b\}$$

```
TESLA_WITHIN(main, previously(
  EVP_VerifyFinal(ANY(ptr), ANY(ptr), ANY(int),
      ANY(ptr)) == 1));
```

Figure 6: A temporal assertion in `libfetch` can describe the certificate verification behaviour expected of `libssl`.

$$\forall b_j \in b \,.\, \forall a_i, a_k \in a \; (a_i \xrightarrow{e_1} a_k \text{ implies } a_i b_j \xrightarrow{e_1} a_k b_j)$$
$$\forall a_i \in a \,.\, \forall b_j, b_k \in b \; (b_j \xrightarrow{e_2} b_k \text{ implies } a_i b_j \xrightarrow{e_2} a_i b_k)$$

3.4.3 Modifiers

TESLA also defines modifiers that control the interpretation of events and operations, including the `optional` modifier and `caller` / `callee`, which control the context in which function instrumentation is added to the program.

3.5 Describing use cases

We now revisit the use cases introduced in section 2, showing how we wrote TESLA descriptions of the intended program behaviour. The interpretation, use and evaluation of these descriptions is described in later sections; here we concentrate on the expressiveness of TESLA and its ability to describe expected behaviour.

For all of these use cases, it is possible to build an ad-hoc mechanism to check the desired property. Such a mechanism might require touching many parts of the source code. It would be *viscous* and *diffuse*, containing *hidden dependencies* and prone to *action slips*, to use Green's Cognitive Dimensions of Notations [15].

In contrast, TESLA provides a general framework that can be directed from a single programmer annotation, written at a level of abstraction appropriate to the property being described. This allows programmers to focus on correctly describing their logic, rather than implementing mechanisms for checking it (and potentially introducing more bugs).

3.5.1 OpenSSL API correctness

In section 2.1, we used OpenSSL as an example of an API that is easy to use incorrectly, creating security risks. A vulnerability was caused by applications failing to properly check tri-state return values. To exploit it, we modified the OpenSSL server to maliciously craft a key-exchange signature that would cause an exceptional failure in a code location that — as of OpenSSL 0.9.8 — used one of these incorrect checks. We did this by forging an ASN.1 tag inside a DSA signature so that one of two large integers claimed to have the `BIT STRING` type rather than `INTEGER`. This caused an exceptional failure inside OpenSSL's `libcrypto` that was incorrectly conflated with success by `libssl` client code.

To demonstrate how TESLA can help discover this kind of logical error, we wrote a simple client that retrieved an HTML document from our malicious `s_server`. This client uses `libfetch`, which uses OpenSSL's `libssl`, which contains the code that calls `libcrypto` incorrectly.

Consider a scenario whereby, on the day after CVE-2008-5077 was announced, the author of this `libfetch` client was curious as to whether or not the client was vulnerable to this kind of problem. Rather than manually inspecting or instrumenting all code that might call `libcrypto` incorrectly, the author can write a TESLA assertion such as figure 6 and then recompile the program and its dependencies. This assertion does not capture the same bug as figure 2: that would be detected by writing an assertion right next to the error itself, which is not a clear demonstration of TESLA's power. Instead, figure 6 shows an assertion in one library (`libfetch`) that can drive instrumentation on either side of another library API (between OpenSSL's `libssl` and `libcrypto`).

The assertion in figure 6 states that, within the context of the `main` execution, a call to `EVP_VerifyFinal` previously returned success. The return value may not have been correctly checked, but if the function returns non-success, it will not satisfy the TESLA expression.

3.5.2 Kernel security framework correctness

We annotated the FreeBSD kernel with 84 assertions documenting 37 inter-process security properties and 47 Mandatory Access Control (MAC) properties. This proved an interesting case study as we had previously implemented these aspects of the kernel [35]. Writing initial assertions took roughly four hours, but debugging the assertions and implementation took an additional two days as we both found bugs in the assertions (reflecting our own misunderstanding of properties of the implementation!) and bugs in MAC itself.

`previously` quantifiers are common in these assertions, which are frequently placed within object implementations (e.g., specific filesystems) but refer to checks in higher-level frameworks (e.g., the Virtual File System (VFS)). We assert that, when an object is accessed within a service, access-control checks have already occurred with the correct subject, object, and parameters. For instance, in protocol-specific socket code (e.g., `sopoll_generic`) we added TESLA assertions stating that mandatory access control checks (in this case, `mac_socket_check_poll`) had previously been performed by central, protocol-agnostic code.

Using these assertions, we found that the MAC check `mac_socket_check_poll` was being invoked for the `select` and `poll` system calls, but not `kqueue`. More subtly, we discovered that one of two present checks was performed using the wrong credential: in `sopoll_generic`, we asserted that MAC must be checked with the `active_cred` credential, but an error in one dynamic call graph caused the cached `file_cred` to be passed down instead of `active_cred`. This error could lead to authorisation being performed using the credential that created the associated file or socket rather than those of the current (active) thread.

```
static int ufs_open(struct vop_open_args *ap) {
  TESLA_SYSCALL_PREVIOUSLY(
    mac_kld_check_load(... vp) == 0
    || mac_vnode_check_exec(... vp ...) == 0
    || mac_vnode_check_open(... vp ...) == 0);
// ...
static int ffs_read(struct vop_read_args *ap) {
  TESLA_SYSCALL(incallstack(ufs_readdir)
    || previously(called(
      vn_rdwr(vp ... flags(IO_NOMACCHECK) ...)))
    || previously(
      mac_vnode_check_read(... vp) == 0));
// ...
```

Figure 7: The UFS implementations of `read` and `open` have code-path–dependent expectations for access-control.

We also make use of `eventually` quantifiers to ensure that necessary security side effects always occur—for example, if a process credential is modified, then the `P_SUGID` process flag must be set to prevent privilege escalation attacks via debuggers. TESLA assertions cover bounded periods of time, with the majority of our security assertions within the lifetime of a single system call (`syscall`). However, we are concerned with certain other cases, such as file-system I/O initiated by virtual-memory page faults (`trap_pfault`).

There is considerable subtlety in the placement and content of TESLA assertions. It took several iterations to develop the checks illustrated in figure 7, which validate that across both system-call and page-fault paths, proper access control takes place. We initially believed that `mac_vnode_check_open` authorised all file-system level open operations, and quickly discovered that different checks handled other open-like operations: loading of kernel modules and execution of binaries. Likewise, file-system reads initiated using the file-system independent `vn_rdwr` may be used "internally" and have MAC checks disabled by `IO_NOMACCHECK`, in which case checks should not be expected by TESLA. One additional instance of `ufs_readdir` occurs within the file system without passing back through VFS. Similar complex structures exist around extended attributes, which may be accessed via system calls, as well as by UFS itself in implementing access-control lists, requiring different enforcement depending on the code path.

As a dynamic system, TESLA relies on test suites and exercise tools (such as fuzzers) to trigger coverage of pertinent code paths—a significant limitation relative to static techniques. However, TESLA itself can help test and therefore improve test coverage: of the 37 inter-process access-control assertions we wrote, 26 were not exercised by FreeBSD's inter-process access-control test suite. Most omissions (19) were in procfs—a deprecated facility disabled by default. Two, however, were in the CPUSET facility added after the test suite was written; five further unexercised assertions were in the POSIX real-time scheduling facility.

```
TESLA_WITHIN(startDrawing, previously(ATLEAST(0,
#include "TESLAGOps.h"
  [ANY(id) push],
  [ANY(id) pop],
  // ...
  [ANY(id) drawWithFrame: ANY(NSRect) inView//...
)));
```

Figure 8: A TESLA assertion that causes instrumentation to be generated for GNUstep GUI debugging.

All of these assertions are subtle and non-trivial, but that is a property of the system they are describing: OS kernels are complex, subtle, and optimised for performance rather than formal analysis. Each TESLA assertion is itself quite succinct, allowing us—in just a few hours of interactive dialogue with the tool—to describe properties that had never been formalised or even documented, let alone proven.

3.5.3 Stateful API exploration

To exploit TESLA's dynamic introspection capabilities, we investigated the GNUstep UI bugs described in section 2.3.

We used TESLA to insert instrumentation and call custom handler code in order to understand the system's dynamic behaviour. Our automata were simple, stating that in between two instrumentation points, which we placed at the start and end of a run-loop iteration, some (or none) of the API methods should have been called. Figure 8 shows the assertion that generates all of the tracing information for this investigation. The `TESLAGOps.h` file contains macros describing the names and types for all (roughly 110) methods that we wish to instrument. The methods listed at the end are those that we wanted to get extra events on method return. The instrumenter generates state-machine events for each call that are handled by both TESLA's automata-checking logic and to be passed to our custom handler.

The cursor push/pop bug was first reported on the GNUstep mailing lists in June 2013, but the causes were not correctly identified. With TESLA, we were able to instrument the library to provide a stack trace every time a `push` or `pop` message was sent and log detailed information about the events being delivered. We provided traces generated by TESLA to the developers at the start of October 2013, allowing them to immediately determine the problem and correctly fix it within a week. Examining the trace, we discovered that mouse-entered events were, in some cases, not correctly paired with mouse-exited events and so the same cursors were pushed onto the cursor stack multiple times. The stack traces that we recorded showed that events invalidating cursor tracking rectangles were being delivered after events that inspected those rectangles. This resulted in a later `pop`

only popping one of a number of duplicated copies of the same cursor, leaving the UI in the wrong state.

The second bug—incorrect output from a new graphics back-end—had already been fixed when we investigated it; our goal was to determine whether finding bugs of this nature was easier with TESLA than with manual inspection. This involved instrumenting around 110 methods, some in the back end and some in the library. This would have been a huge undertaking to do manually. With TESLA, we were able to generate detailed event traces, describing exactly which view class was responsible for calling each back-end method and to understand. The bug was caused by the new back end's inability to save and restore graphics states in a non-LIFO order. This was caused by the author of the code not being aware that this was a valid sequence of operations, something obvious in traces of even simple application.

Investigating both bugs required a few hundred lines of changes in three files. Of these, half were in the `TESLAGOps.h` file created simply to list the selectors that we wished to instrument. Most of the remainder was code for formatting the traces. This modification let us track complex interaction between two software libraries and uncover the source of subtle interaction bugs between loosely coupled components.

By understanding the flow of execution through a complex library, it also becomes possible to make more informed decisions about how it should evolve. These traces make it possible to examine common sequences of operations, exposing potential optimisation opportunities. This is difficult to discover statically, as many views delegate drawing to 'cells' (simple classes that draw data in a particular way) that are provided by another object. For example, according to our profiling, applications often save and restore the graphics state (a comparatively expensive operation), when the only aspects of the state that are changed in between are the current drawing location and the colour. In some complex views, the restore is unnecessary, because the next cell always explicitly sets these values and so these calls might be elided. This would require invasive changes to the internal APIs, but before examining these traces it was not obvious that this would be a worthwhile change.

4. TESLA implementation

TESLA uses the LLVM compiler construction libraries [22] and the associated C front-end Clang [21]. The TESLA workflow uses three components: a Clang-based analyser to parse automata from C-based languages, an LLVM-based instrumenter to insert event hooks into the LLVM IR (intermediate representation), and a runtime support library that tracks automata instances and their state.

4.1 Analyser

The TESLA analyser uses the Clang Tooling framework [25] to consume C abstract syntax trees and parse assertions contained in them. Since TESLA uses the Clang front-end for its analysis, it benefits from the same syntax- and type-checking, scoping rules, etc. as a normal compilation pass.

The TESLA analyser performs a recursive descent over an abstract syntax tree (AST) constructed by Clang, parsing the expressions it finds and converting them into automata states and transitions. For instance, the assertion `TESLA_WITHIN(syscall, eventually(foo(x)==0))` will be translated into an automaton with a chain of transitions driven by events `call(syscall)`, `TESLA_ASSERTION_SITE`, `foo(x)==0`, and `returnfrom(syscall)`, as well as bypass `returnfrom(syscall)` transitions to allow code paths that call `foo` but never pass through the assertion site.

Using Clang, we could also support other C-based languages such as C++ and Objective-C. We do not currently support C++, as we primarily target C-based trusted computing bases (TCBs), but we do support assertions that use Objective-C. Parsed assertions are converted into an automaton representation, stored on disk in a file with a `.tesla` extension and formatted using Google Protocol Buffers [13].

The assertions in every file can name events defined in any other file, so we combine `.tesla` files into a larger file describing all parts of the program that may need instrumentation. This interdependency complicates iteratively rebuilding a program: when one C file changes, it changes the combined `.tesla` file. This causes re-instrumentation of all LLVM IR files; the impact of this is discussed in section 5.1.

4.2 Instrumenter

The TESLA instrumenter modifies compiled code to turn program events into automaton transitions, transforming LLVM IR generated by language front-ends. We currently focus on C and Objective-C produced by Clang, but can instrument IR generated from any source language.

LLVM IR is a typed assembly language in static single assignment (SSA) form for an abstract machine with an infinite register set whose contents may only be assigned once. This abstract representation is convenient for instrumentation because register contents are immutable.

Instrumentation is not robust in the presence of function inlining and other optimisations, so we run the TESLA instrumenter before optimisation, i.e. we run the Clang front-end with `-O0`, instrument its output and then run the LLVM optimiser `opt` with `-O2` on the result.

The TESLA instrumenter adds two kinds of code during program instrumentation: *program hooks* that identify program events and *event translators* that match events and convert them to automata symbols. Programs hooks are simply calls to generated functions, and they can be inlined in some cases (when using caller-side instrumentation or when employing link-time optimisations). They expose events including function call/return, structure field assignment and assertions to the event translators described below.

Function call/return TESLA can instrument function calls and returns in either callee or caller context. The former is desirable when instrumenting functions defined by the pro-

grammer, whereas the latter is important when instrumenting calls into a library that cannot be recompiled. Callee instrumentation adds instrumentation to the target function's entry basic block and before any return instructions. Caller instrumentation is inserted immediately before and after a call site.

Field assignment The same approach is taken for the direct assignment to structure fields. Unlike function events, there is no callee context: the code that modifies the structure field is the code that must be modified. The event translator for a field assignment takes as arguments the structure containing the field, a pointer to the field (and thus its current value) and the new value that is being assigned.

Assertions The program must also be transformed so that, on reaching TESLA assertion sites, the call to the TESLA pseudo-function `__tesla_inline_assertion` is replaced by a call to an appropriate event translator. The values of variables named in the assertion are taken from the local scope and passed to the event translator and the original call to the unimplemented `__tesla_inline_assertion` is removed.

Event translators The instrumenter generates event translators that convert program events into automata symbols. Event translators are generated as chains of basic blocks, with two tasks per automaton that references the event.

First, the generated code checks static event parameters. For instance, in the assertion `eventually(foo(x)==0)`, the event should only be reported if `foo(x)` returned `0`. Otherwise, the translator branches to the static checks for the next automaton. Second, if the static checks passed, it allocates a fixed-size data structure on the stack, populates it with the dynamic variable–value mapping and passes it to `libtesla`'s `tesla_update_state` function. In the simple example above, `libtesla` must check not just that the `foo(x)==0` event occurred but whether the value of `x` matches that which was previously encountered in the assertion's scope.

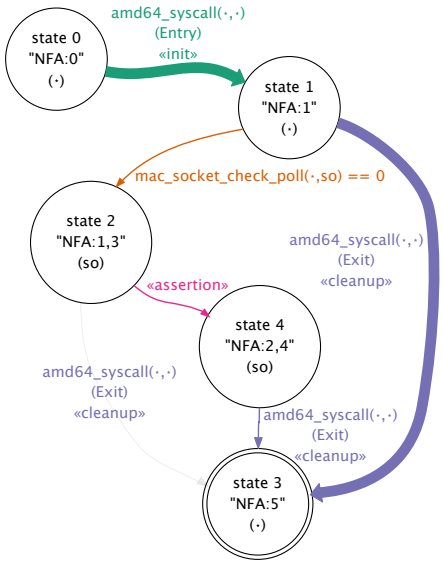
The only conditional control flow within event translators is the ability to skip this second task if the checks in the first task fail; in all cases, the generated code will make forward progress under the assumption that our hand-written `tesla_update_state` function makes forward progress.

4.3 Objective-C instrumentation

Instrumenting a dynamic language presents additional challenges. In C, a call to a function is static flow control, with a destination that is unique. In Objective-C, interprocedural flow control is either a C function call or a message send; methods can be replaced at run time, so even for an object of a known class it is impossible to tell statically which method will be invoked for a given message send.

Fortunately, dynamic behaviour also provides an opportunity: message sends are implemented by the `objc_msgSend` function, provided by the Objective-C runtime library. We modified these functions in the GNUstep Objective-C runtime [10] to provide a new interposition mechanism.

Before calling any method, the runtime consults a global table of interposition hooks and determine whether there is a hook for the specified selector. This mechanism allows us to provide callee-side instrumentation without access to source code, but it does impose a performance penalty.




```
TESLA_SYSCALL_PREVIOUSLY(
      mac_socket_check_poll(ANY(ptr), so)==0);
```

Figure 9: An automaton for a MAC check assertion. Transitions are weighted according to their occurrence at run time.

4.4 libtesla

`libtesla` is the run-time support library for TESLA. It accepts streams of events and uses them to manage automata instances. Most of `libtesla` is written in portable C. Around 11% is specific to the FreeBSD kernel environment, half of which is specific to DTrace.

`libtesla` can store automata state in either a global or a thread-local store, as specified by the programmer (see section 3.2). Each store can hold a number of automata *classes*—one for each programmer-specified automaton—that can each be *instantiated* a number of times, differentiated by the variables they reference. For example, to implement the assertion `previously(foo(x)==0)`, `libtesla` must instantiate one instance of the automaton for each value of `x` for which a `foo(x)==0` event is observed.

4.4.1 The lifetime of an automaton instance

Figure 9 illustrates an automaton class, derived from a MAC assertion, which can be instantiated by `libtesla`.

Init Automata instances are created with an «init» transition; the example in figure 9 begins by entering FreeBSD's `amd64_syscall` function. This transition creates an instance in state 1 with name $(*)$: although the assertion refers to a variable `vp`, at the point of entry into the system call it is not yet known what the relevant value of `vp` will be.

Clone After a successful call to `mac_vnode_check_read`, an instrumentation hook will forward the third parameter to

`libtesla`, which must then **clone** a copy of the $(*)$ automaton in state 1 into an automaton named $(\texttt{vp}_1)$ in state 2. If another call is made to `mac_vnode_check_read` with different arguments, `libtesla` will clone another copy of the $(*)$ automaton, yielding an automaton named $(\texttt{vp}_2)$ in state 2.

Update At this point, if the assertion site is reached with local variable `vp` equal to $\texttt{vp}_1$ or $\texttt{vp}_2$, the correspondingly-named automaton instance will be updated to state 4.

Error If, after the above transitions, the assertion site is reached with variable $\texttt{vp} = \texttt{vp}_3$, no instance can be found to update: the function `mac_vnode_check_read` has *not* been called with parameter $\texttt{vp}_3$ and successfully returned, so an error will be reported (see section 4.4.2).

Cleanup Finally, a «cleanup» transition resets an automaton class: all instances are expunged and `libtesla` resumes ignoring events until the next «init». In the kernel we rely on preallocation to avoid dynamic allocation in code paths that do no permit it (e.g., while holding mutexes). Bounds such as "within the current system call" allow this strategy: we preallocate a fixed-size memory block per thread, giving a deterministic memory footprint, and report overflows so that we can adjust preallocation size on the next run.

4.4.2 Dynamic introspection

TESLA has a pluggable event notification framework with a set of default handlers and support for user-provided handler callbacks. `libtesla` reports all of the event types referenced in section 4.4.1: instance initialisation, clones, updates, errors, and finalisation (automaton acceptance).

In userspace, TESLA's default behaviour is to output event information to `stderr`, controlled by the `TESLA_DEBUG` environment variable. Mismatches between temporal specifications and actual behaviour cause the program to fail-stop by default, but this is configurable at run-time. In the FreeBSD kernel, the default handler uses DTrace [7] to aggregate information across events, e.g., counting how often a transition is triggered per stack trace.

TESLA can combine observations of dynamic behaviour with static automata descriptions, producing weighted graphs like that in figure 9. This allows the programmer to visually inspect the portions of the state graph that are executed in practice, as well as their relative frequencies. This visibility can be used to test tool development, like traditional code coverage analysis but at a logical rather than source-line or machine-instruction level.

5. Performance

We have shown that TESLA is able to describe many useful temporal behaviours, detect deviations from programmer assertions, and help programmers understand run-time behaviour. We now evaluate TESLA's performance for the use cases described in section 2. TESLA imposes performance overhead on both program build and run-time behaviour. In most cases, these overheads are similar to costs accepted in systems communities that use dynamic development tools.

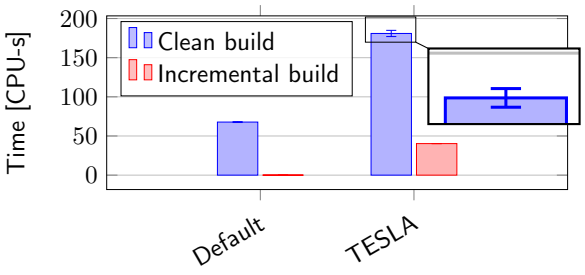


Figure 10: The TESLA toolchain slows down the OpenSSL build process, especially when rebuilding incrementally.

5.1 OpenSSL API correctness

The OpenSSL API property that we describe in section 3.5.1 ("previously, the key-exchange signature was successfully verified") is a one-shot event: applications call `SSL_connect` at the beginning of a connection, not repeatedly in a tight loop. The performance impact on this simple case study is encountered at build time.

When building software with TESLA assertions and instrumentation, there are additional stages in the compilation workflow, so build times can increase by as much as $2.5\times$, as shown in figure 10. The real cost of the TESLA workflow, however, is in incremental *rebuilds*.

TESLA assertions in any source file can reference events that are defined in any other source file. For instance, in our case study, we wrote an assertion in the client's `main` function that references a call in OpenSSL's `ssl3_get_key_exchange` function to `EVP_VerifyFinal`. Depending on whether caller- or callee-side instrumentation is preferred, instrumentation may need to be added when compiling either `ssl/s3_clnt.c` or `crypto/evp/p_verify.c`.

The practical consequence is that, after modifying a TESLA assertion in any one source file, instrumentation must be performed again, potentially on many files. In our current implementation, we naively re-instrument all code, leading to the approximately $500\times$ incremental slowdown shown in figure 10. This process could be pared down through further build optimisation, but the one-to-many nature of re-instrumentation is a fundamental problem.

In practical terms, an increase from near-instantaneous incremental rebuilds to approximately 30s is a burden for developers, but that cost must be weighed against the benefit of identifying correctness issues that are difficult to detect with conventional tools.

5.2 Kernel security framework correctness

Our use of TESLA in the FreeBSD kernel—introduced in section 3.5.2—uncovered five functionality bugs with subtle security implications, and substantially enhanced our understanding of several kernel properties. This benefit came at

Symbol	Description	Assertions
M_F	MAC (filesystem)	25
M_S	MAC (sockets)	11
M_P	MAC (processes)	10
M	All MAC assertions	48
P	Process lifetimes	37
All	All TESLA assertions	96

Table 1: Assertion sets referenced in figure 11.

a cost of less than $1.35\times$ when running under our instrumented kernel, which is somewhat slower than other debugging tools accepted by the FreeBSD developer community in the command case, but which is perfectly acceptable for developers of subsystems or for regression test infrastructure.

5.2.1 Build-time overhead

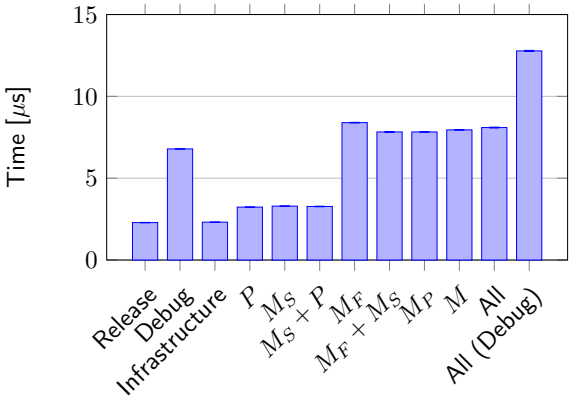
Instrumenting the kernel requires changing the build process to emit LLVM IR on the path to object code. When we add TESLA analysis and instrumentation to the LLVM bitcode workflow, from-scratch build times increase with the number and complexity of rules, taking up to $2.2\times$ longer to build and link with our set of TESLA assertions. This is a comparable slowdown to the OpenSSL case study. Incremental rebuild of an instrumented kernel with no assertions takes $3.5\times$ longer, and a kernel with 85 assertions takes $37\times$—only a modest 30% savings vs. a clean build.

5.2.2 Run-time overhead

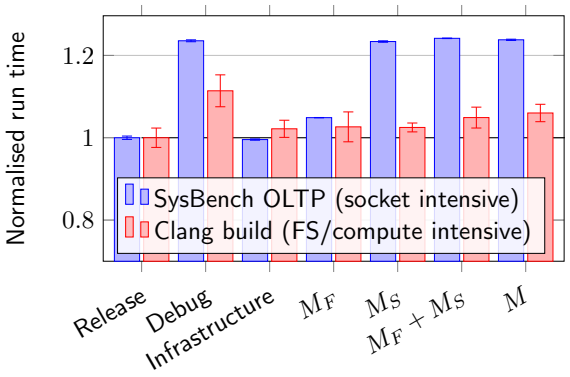
The FreeBSD community tolerates a number of performance impacting development aids in the development branch. For example the WITNESS and INVARIANTS kernel options cause up to a 15% slow down in the macrobenchmarks discussed below and up to a $3\times$ slowdown in microbenchmarks. The enabling of these options by default demonstrates that developers and other users of development trees are typically willing to accept these overheads.

To determine the practical overhead of TESLA kernel assertions we ran benchmarks on a kernel in a release configuration, one with standard debugging options including WITNESS and INVARIANTS, and several TESLA-instrumented kernels. The kernel was from the FreeBSD 10-CURRENT development branch on June 3 2013 (svn r229293) with (conditionally compiled) TESLA modifications applied. The variants of TESLA-instrumented kernels included one with just the instrumentation framework and test assertions enabled ("Infrastructure"), various combinations of the assertion sets listed in table 1, one with all TESLA assertions enabled ("All") and one with all assertions enabled in addition to normal FreeBSD debugging features ("All (Debug)").

As a typical developer workload we benchmarked the time required to perform a compile of the Clang-3.3 compiler in all these configurations. We also ran the SysBench [20] 0.4.12 OLTP benchmark against a MySQL



(a) A system-call–intensive microbenchmark (the lmbench suite's `open close`) is measurably slowed by TESLA.



(b) TESLA's impact on larger workloads is comparable to existing debugging tools and proportional to instrumentation encountered.

Figure 11: The performance impact of kernel TESLA assertions on microbenchmarks and larger code.

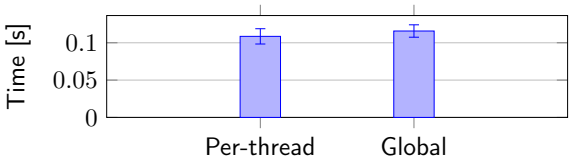
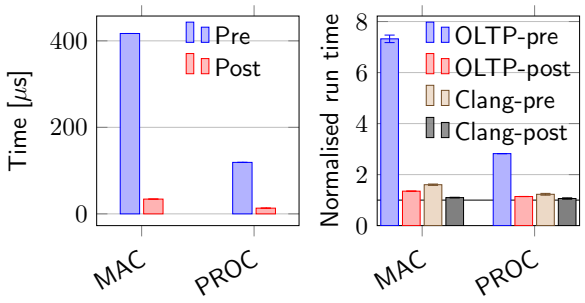


Figure 12: Global assertions require explicit synchronisation, which comes at a run-time cost.

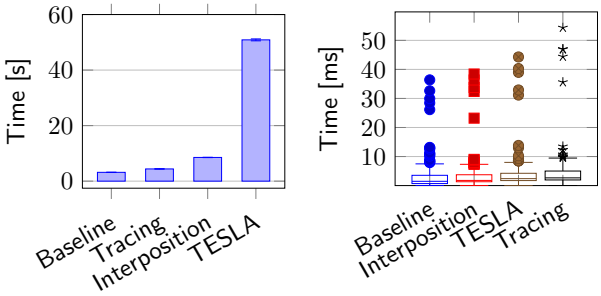
5.6.14 database as an example of a transaction sensitive workload. In addition to these macrobenchmarks we ran portions of the lmbench [29] microbenchmark suite against the same set of kernels. All FreeBSD benchmarks were run on servers with an Intel E5-1620 (SandyBridge) 3.60GHz CPU, 64GB of RAM, and a 500GB SSD running a FreeBSD amd64 9.2-RELEASE userland. The MySQL database was memory backed and builds were performed on the SSD.

Almost all of our assertions were in the thread-local context, so they could take advantage of an existing event serialisation. The cost of a TESLA-imposed serialisation is shown



(a) Microbenchmark (b) Macrobenchmark

Figure 13: Performance improvements with optimisation.



(a) Microbenchmark. (b) Macrobenchmark.

Figure 14: TESLA has a significant impact on Objective-C messages but little impact on user-perceived performance.

in figure 12. This serialisation is lock-based, so contention would increase the cost further.

Our first, naive implementation caused a slowdown of almost 2× in Clang builds and 10× in the OLTP benchmark. Developers may occasionally accept high overheads when running e.g., debug LLVM builds or under `valgrind`), but not routinely as with FreeBSD debug features.

This performance impact was caused by many assertions sharing the same *temporal bounds*: on entering a system call, `libtesla` would do work on every system-call–related automaton. We optimised for this common case by keeping a per-context (global or per-thread, see section 3.2) record of common initialisation and cleanup events and doing *lazy initialisation* of automaton instances after they received their first non-initialisation event. As shown in figure 13, microbenchmark results improved dramatically from nearly 100× slowdown to less than 7× and Clang builds were reduced to less than 10% overhead — comparable to normal FreeBSD debug features WITNESS and INVARIANTS.

TESLA's run-time overhead is still measurably higher than other development and debugging tools accepted by the FreeBSD project. Nonetheless, we expect that many developers of systems using TESLA would only run with a subset of assertions enabled—which assertions would depend on what they are working on—and that automated testing infrastructure would run test suites with all assertions enabled.

5.3 State machine exploration

For this case study, all our TESLA declarations are within a single compilation unit. Although instrumentation spans two libraries and multiple classes, it is all inserted via interposition and so we only need to run the instrumenter on a single compilation unit. This means that incremental builds still work, as do parallel builds, and so TESLA has a negligible impact on build times.

5.3.1 Run-time overhead

The effect of TESLA instrumentation on sending Objective-C messages is illustrated in figure 14a. This figure shows the time spent in a tight message-sending loop — without doing any other work — in a normal release build, a build linked against the Objective-C runtime with tracing enabled, with tracing enabled and a trivial interposition function on the message send, and with a TESLA automaton processing the events, where the run time is up to 16× longer. As in the FreeBSD case, the microbenchmark shows a very high overhead, but a more practical use of TESLA (our exploration of AppKit API usage) reveals a more reasonable overhead. We used GNU Xnee [33] to replay X11 events and interact with dialog boxes, and figure 14b shows window redrawing times: the majority of events only repaint portions of the window, and outliers are complete redraws.

The four modes in this figure are without tracing support compiled into the Objective-C runtime library, with tracing support but without TESLA using it, with TESLA enabled and monitoring the events, and with custom event handlers enabled. When running with all of our tracing enabled, the longest redraw is 54ms — allowing smooth animation — and most redraws are well under 10ms. This is fast enough that TESLA instrumentation could be deployed in a release build, although trace generation would likely be disabled.

6. Related work

Temporal properties Chen et al.'s MOPS [8, 9] also represents temporal properties as automata and uses static model checking to determine if the property is satisfied. However, their expressivity is quite limited as automata transitions describe only function calls, but do not include program state (e.g., pointer arguments and return values). Similar limitations also apply to later work in aspect-oriented debugging: languages such as AWED [30] and SPoX [16] are limited to static descriptions of program events. In contrast, TESLA's automaton language can describe events that are parameterised by dynamic variable–value mappings.

Weimer and Necula have worked on mining temporal specifications from Java programs [36]: they can extract specifications of two-state automata (e.g., `begin`/`commit` loops) from Java code by statically examining exception-handling code paths. In contrast, our approach is to enable

programmers to directly describe complex high-level properties and mechanically check them at run-time.

Lawall et al.'s Coccinelle [23] addresses the problem described in our OpenSSL use case with a text processing tool that has some C-specific knowledge but does not completely capture C expressions. By building on a modular compiler framework, TESLA supports complex language constructs and can keep pace with language development.

Gottschlich et al.'s Concurrent Predicates [14] introduce synchronisation primitives that helps programmers reproduce concurrency bugs: a Concurrent Predicate can be thought of as a barrier conditioned on an arbitrary predicate. This requires programmers to effectively describe local preconditions in the negative ("my code will crash if X"); in contrast, TESLA allows programmers to specify temporal properties across arbitrary time spans and numbers of events.

Static analysis There has been a great deal of work done on static analysis of systems code. Wu et al. have used static AST analysis to find memory leaks in the MAC Framework, a class of localised temporal bug [37]. Ball et al.'s SLAM [2] and Beyer et al.'s Blast [4, 18] utilise *counterexample-guided abstraction refinement*, in which tools progressively refine an abstract code model until they prove the property, find a counterexample, or time out. Cook has demonstrated termination proofs on systems code [11] and Lu et al.'s MUVI [26] detects multi-variable concurrency bugs. In all cases, however, output is not deterministic and run times may be prohibitive: on large programs, MUVA had run times from 20 minutes to three hours, Cook's work on device drivers included runs of over 100,000 seconds, and Blast often gives up after a timeout. TESLA, in contrast, imposes a run-time performance cost, but for the properties it can check, it can always check them on exercised code paths.

Symbolic execution Cadar et al.'s KLEE [6] provides a framework for symbolic execution of LLVM IR. This provides a way of exhaustively testing all possible code paths, but is time consuming. Using KLEE with large code bases such as OS kernels is not currently feasible: the number of potential execution paths becomes unmanageable.

Program instrumentation Cantrill et al.'s DTrace [7] provides a framework for inserting probes into software. By itself, DTrace cannot describe the complex temporal properties that TESLA targets, nor can it instrument structure field assignment. An early TESLA prototype attempted to extend the D language for this purpose, but we abandoned it in favour of a compiler-driven approach. TESLA does use DTrace's static probe functionality to aggregate and report on temporal events in the FreeBSD kernel.

Yanagisawa et al. use the *Kerninst* binary patching framework to implement aspect-oriented programming in the Linux kernel [38]. They are able to run code at arbitrary point cuts, but without compiler-assisted instrumentation, they must generate machine code that interoperates with register allocation, etc. in the surrounding context. Kerninst could also support TESLA, but our integration with the compiler provides greater assurance of correctness and lets us statically check properties of the instrumentation itself. TESLA could also utilise other binary instrumentation systems such as Luk et al.'s PIN [27] and Feiner et al.'s DynamoRIO [12], but their dynamic overhead would add to that of TESLA's automata management.

Another AOP-like is Harris et al.'s Capweave [17], which uses visibly pushdown automata to describe logical and security requirements, driving binary weaving of security-related system calls into a program. This model cannot describe arbitrary high-level events, nor assist the programmer in understanding dynamic behaviour.

Program understanding Lefebvre et al's Tralfamadore [24] provides debugging access to the complete execution trace of an application, allowing temporal problems to be diagnosed. However, it does not provide a programmer with a vocabulary for describing assertions.

7. Future work

TESLA assertions can refer to values in the current scope, but some temporal properties can only be described by binding events together with values that are no longer known. A related problem relates to function pointers: TESLA can instrument functions called by pointer in the caller context but not the callee context because it lacks the vocabulary to describe function pointer assignment when that pointer value is not available in the assertion scope. We intend to introduce free variables in future versions of TESLA, which will support this assertion functionality, using the existing instrumentation and runtime infrastructure.

Currently, `libtesla` allocates its own memory for automata storage. Performance improvements could be gained by allowing users to delegate space within data structures of the instrumented program. This would naturally lead to per-object assertions, allowing assertions to be more easily tied to an object's lifetime. Existing object locks could be used or elided if proven to be held.

We have focused exclusively on dynamic analysis. A natural next direction would be to explore cases where static analysis could be used to both improve accuracy and performance. Where inter-procedural analysis is reliable, and in particular when using per-thread contexts, it might be that otherwise expensive sequences of checks and state transitions could be entirely elided. A further advantage would be compile-time reporting of potential failures.

Although we have performed some optimisation on common runtime cases, there is still further scope for performance refinement. For instance, build time is partly hampered by our tool re-loading, re-parsing, and re-interpreting the same TESLA automaton description for every LLVM IR file it instruments. Build-time overheads are also based on a conservative compilation strategy: whenever any assertion is modified, all LLVM IR files are re-instrumented us-

ing the new automata descriptions. This strategy is overly pessimistic, as there are cases in which only limited re-instrumentation is necessary. For instance, if an automaton is rearranged or removed without introducing new instrumentation points, TESLA need only re-instrument files that are known to contain instrumentation points.

Extending the system to support C++ may also be of interest, but is beyond the scope of the current evaluation.

Allowing post-processing of event traces from tools such as Tralfamadore [24] would offer an alternative, non-instrumentation–based checking scheme with different performance vs. timeliness properties.

8. Lessons learned

When we began this work, we expected false positives and false negatives, common to C-based static and dynamic analysis, to cause significant problems. Surprisingly, none of the complex examples we present encounter this problem, partly because TESLA warns programmers about unsafe code that would violate TESLA's program model. Taking data outside the type system (e.g., casting it to `char*` and mutating raw bytes) would cause TESLA to miss program events, but we worked with software that did not take this approach.

Placing the tracing events on interfaces between modular code has a low overhead (compare figure 14a and figure 14b), but does not always catch bugs completely isolated within a component. TESLA complements rather than replaces unit testing, which is good at finding localised bugs.

The interposition mechanisms available in dynamic languages make generating TESLA call instrumentation much simpler, allowing instrumentation without recompiling every caller or callee. This could be translated to C if we added a compilation mode that guaranteed every call a dynamic relocation and a run-time linker that allowed interposition.

9. Conclusion

TESLA is a dynamic assertion framework for expressing and validating the complex temporal properties key to the correctness of large-scale software systems. We have demonstrated TESLA through three case studies: OpenSSL, the FreeBSD kernel, and the GNUstep user interface, illustrating API use checking, validation of temporal security properties, and functional coverage checking in test suites. In addition to checking temporal properties, TESLA enables exploration of runtime behaviour, including visualising automata transitions and profiling event frequencies. Through integration with DTrace, programmers can gain new insight into the behaviour of complex systems. TESLA's performance impact is comparable with accepted debugging overheads for target development communities, but also experiences performance proportionality—developers pay for the assertions they enable. We have focused on programmer-friendly and practical dynamic analysis and instrumentation, but TESLA also enables creation of a new corpus of semantically rich and machine-readable assertions to enable increasingly powerful static analysis and formal verification tools.

Acknowledgments

We thank our colleagues for their helpful comments, including Ben Laurie, Steven Hand, Anil Madhavapeddy, Peter G. Neumann, Hassen Saidi and Steven Murdoch, as well as anonymous reviewers and our shepherd, Dejan Kostic. Portions of this work were sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense. We gratefully acknowledge Google, Inc. for its sponsorship.

References

[1] J. H. Baldwin. Locking in the Multithreaded FreeBSD Kernel. In *Proceedings of the BSD Conference 2002*. USENIX Association, 2002.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Jan. 2002.

[3] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report MTR-2547, Nov. 1973.

[4] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, Sept. 2007.

[5] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE Corporation, June 1975.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004.

[8] H. Chen, D. Dean, and D. Wagner. Model Checking One Million Lines of C Code. In *NDSS 2004: Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 171–185, 2004.

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

[10] D. Chisnall. A new Objective-C runtime: from research to production. *Communications of the ACM*, 55(9), Sept. 2012.

[11] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 415–426. ACM, June 2006.

[12] P. Feiner, A. D. Brown, and A. Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *ASPLOS '12: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–146. ACM Press, 2012.

[13] Google. Google Protocol Buffers. URL: `https://developers.google.com/protocol-buffers/`.

[14] J. Gottschlich, G. Pokam, C. Pereira, and Y. Wu. Concurrent predicates: A debugging technique for every parallel programmer. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 331–340. IEEE Press, 2013.

[15] T. Green. Cognitive Dimensions of Notations. In A. Sutcliffe and L. Macaulay, editors, *People and Computers V*, pages 443–460. Cambridge University Press, 1989.

[16] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*. ACM, June 2008.

[17] W. R. Harris, S. Jha, T. Reps, J. Anderson, and R. N. M. Watson. Declarative, Temporal, and Practical Programming with Capabilities. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*, pages 18–32, May 2013.

[18] T. A. Henzinger, G. C. Necula, R. Jhala, G. Sutre, R. Majumdar, and W. Weimer. Temporal-Safety Proofs for Systems Code. In *CAV 2002: Proceedings of the 2002 Conference on Computer-Aided Verification*, pages 526–538, Sept. 2002.

[19] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *SOSP '09: Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*. ACM, Oct. 2009.

[20] A. Kopytov. SysBench: a system performance benchmark. URL: `http://sysbench.sourceforge.net`.

[21] C. Lattner. LLVM and Clang: Next Generation Compiler Technology. In *BSDCan 2008*, 2008.

[22] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04)*, pages 75–86, 2004.

[23] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller. Finding Error Handling Bugs in OpenSSL Using Coccinelle. In *Dependable Computing Conference (EDCC), 2010 European*, pages 191–196, 2010.

[24] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore: unifying source code and execution experience. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 199–204, New York, NY, USA, 2009. ACM.

[25] LLVM. Clang 3.4 documentation: LIBTOOLING. URL: `http://clang.llvm.org/docs/LibTooling.html`.

[26] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP '07: Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. ACM, Oct. 2007.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200. ACM, June 2005.

[28] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, Aug. 2004.

[29] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[30] L. D. B. Navarro, R. Douence, and M. Südholt. Debugging and Testing Middleware with Aspect-Based Control-Flow and Causal Patterns. In *Middleware 2008*, pages 183–202. Springer, 2008.

[31] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, June 2007.

[32] OpenSSL. OpenSSL security advisory (07-jan-2009). URL: `http://www.openssl.org/news/secadv_20090107.txt`.

[33] H. Sandklef. GNU Xnee. URL: `http://www.sandklef.com/xnee/`.

[34] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*, June 1986.

[35] R. N. M. Watson. A decade of OS access-control extensibility. *Communications of the ACM*, 56(2), Feb. 2013.

[36] W. Weimer and G. C. Necula. Mining Temporal Specifications for Error Detection. In *TACAS 2005: Proceedings of the Eleventh International Conference on Tools and Algorithms for the Construction and Analyis of Systems*, pages 461–476. Springer, 2005.

[37] X. Wu, Z. Zhou, Y. He, and H. Liang. Static Analysis of a Class of Memory Leaks in TrustedBSD MAC Framework. In *Proceedings of the 2009 International Conference on Information Security Practice and Experience*, pages 83–92. Springer, 2009.

[38] Y. Yanagisawa, K. Kourai, and S. Chiba. A dynamic aspect-oriented system for OS kernels. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. ACM, Oct. 2006.