IBOS: A Correct-By-Construction Modular Browser*

Ralf Sasse[1], Samuel T. King[2], José Meseguer[2], and Shuo Tang[2]

[1] Institute of Information Security, ETH Zurich, Switzerland
ralf.sasse@inf.ethz.ch
[2] University of Illinois at Urbana-Champaign, USA.
{kingst,meseguer,stang6}@illinois.edu

Abstract. Current web browsers are complex, have enormous trusted computing bases, and provide attackers with easy access to computer systems. This makes web browser security a difficult issue that increases in importance as more and more applications move to the web. Our approach for this challenge is to design and build a correct-by-construction web browser, called IBOS, that consists of multiple concurrent components, with a small required trusted computing base. We give a formal specification of the design of this secure-by-construction web browser in rewriting logic. We use formal verification of that specification to prove the desired security properties of the IBOS design, including the address bar correctness and the same-origin policy.

Keywords: Browser security, same-origin policy, rewriting logic.

1 Introduction

The modern web browser has become a popular target for attackers of computer systems [20,11,13,12,16] – two key factors contribute to this trend. First, browsers are complex software artifacts that are riddled with security vulnerabilities. For example, Internet Explorer, Chrome, Safari, Opera, and Firefox had over 500 new security vulnerabilities combined in 2010 [16]. Second, browsers are the primary way users access the wide array of current web-based applications. Web-based applications are collections of web pages that people use in concert to carry out common computing tasks. As users continue to use browsers for more fundamental computing needs, the browser itself contains more valuable data, such as banking credentials, login information, and credit card numbers, presenting an enticing target for attackers of computer systems.

Current research efforts into more secure web browsers help to deal with the complexity of browsers themselves by decomposing them into smaller components. The OP web browser [7], Gazelle [19], Chrome [2], and ChromeOS [6] propose new browser architectures for separating the functionality of the browser

* This work was done while the first author was at the University of Illinois.

from security mechanisms and policies. This privilege separated architecture enables a small program, called a *browser kernel,* to enforce browser security policies without relying on the correct operation of the millions of lines of code used to implement the browser. In addition to these alternative browser architectures, the IBOS system [17] extends these modularity principles to the operating system to remove almost all traditional operating system (OS) components and services from the browser's trusted computing base (TCB).

In this work we first present the design of IBOS, which is highly modular, with the browser kernel being separated from all other processes. Our presentation includes a discussion of the browser's security goals, in particular the ability to enforce security policies, like the same-origin policy (SOP), and a trusted user interface, to prevent address bar manipulation. The browser is also resilient against having some of its components subverted, due to its modular structure and central trust in the kernel only.

We give a formal specification of IBOS in rewriting logic [10], showing its modular structure and its communication paths. The formal specification is executable in the Maude tool [5], which is a high-performance implementation of rewriting logic. The IBOS security properties mentioned above are then model checked in Maude for suitable bounds. Furthermore, we prove that the bounded model-checking results thus obtained do actually extend to the unbounded case, i.e., the full operation of the browser. In this way we prove that the browser design implements SOP correctly and that its address bar cannot be spoofed, i.e., it will always show the URL for the content on screen. Our analysis did find an easy to correct bug related to how the display memory is handled, which makes browser tabs inoperable, as their content does not update anymore.

Organization. The rest of this paper is organized as follows. Section 2 will show some preliminaries on rewriting logic and the Maude tool. In Section 3 we explain the IBOS system including its design and security properties. Section 4 presents a high-level picture of the formal specification of IBOS. In Section 5 we show the formal verification of the security goals of IBOS. Finally, in Section 6 we discuss related work and present some conclusions.

2 Preliminaries

In this paper, we follow the classical notation and terminology from [18] for term rewriting, and from [10] for rewriting logic. Rewriting logic specifications are rewrite theories, $\mathcal{R} = (\Sigma, E \cup Ax, R)$, with $(\Sigma, E \cup Ax)$ an equational theory. Σ is the set of typed function symbols, sorts, and subsorts, while the equations E together with the axioms Ax specify the set of *states* of $\mathcal{R}$ as an algebraic data type. This equational theory represents the deterministic part of the system, while the rules R represent the concurrent aspects and work on top of that data type. The rewrite theory $\mathcal{R}$ provides both a *mathematical model* and an *executable semantics* by term rewriting.

The *Maude* tool [5] is a high-performance implementation of rewriting logic. It allows equational specification in *functional modules*, corresponding to equa-

tional theories $(\Sigma, E \cup Ax)$, and full rewrite theories $\mathcal{R} = (\Sigma, E \cup Ax, R)$ can be specified as *system modules*. In functional modules other modules can be included, sorts and subsorts can be declared and operator symbols can be defined, possibly with equational attributes (called axioms) like associativity, commutativity and/or identity. Sorts, subsorts, conditional equations and memberships define the computations that are possible. Reasonable executability requirements are needed to make a module *admissible* (see [5, Sections 4.6 and 6.3]), including termination (modulo axioms), ground confluence and sort-decreasingness. Then, Maude can execute the module by equational simplification modulo the axioms, where the equations in E are used as rules from left to right and Maude's built-in matching modulo the axioms Ax leads for each term t to its canonical form with a least sort. For functional modules this yields the algebra of canonical forms $Can_{\Sigma/E \cup Ax}$ which is isomorphic to the initial algebra semantics given by $T_{\Sigma/E \cup Ax}$ (see [5, Sections 4.6-4.8]). Equational simplification modulo axioms is executed by the `reduce` command in Maude. Maude also has built-in support for the modeling of objects, in the `CONFIGURATION` module, which we use here.

In order to be admissible, a system module has to, in addition to its equational component being admissible, satisfy the ground coherence requirement of its rules R with respect to equations in E and also needs to ensure that all variables in rules can be instantiated by (incremental) matching. Such a module can be executed in Maude by rewriting with the rules and oriented equations modulo axioms Ax. This yields an initial reachability model $\mathcal{T}_{\mathcal{R}}$ whose states are elements of $Can_{\Sigma/E \cup Ax}$ and whose transitions are rewrites. Rewrites in a system module are performed in Maude by the `rewrite` command, which is position fair and rule fair. Breadth-first search is also available using the `search` command. A linear temporal logic (LTL) model checker is built-in for verification of safety and liveness properties.

3 Illinois Browser Operating System

We present the formal verification of the design and implementation of an experimental operating system called the Illinois Browser Operating System (IBOS) [17]. IBOS is an operating system and a browser co-designed to drastically reduce the trusted computing base (TCB) for a web browser and to simplify the browsing system. We first give a brief introduction about the background of the state-of-the-art of web browser security, then describe the architecture and design principles of IBOS to show why its component-based design is suitable for formal verification, and finally explain its key security properties related to the browser components that we are going to verify.

3.1 Web browser security background

The web is now the dominant platform for delivering interactive application to hundreds of millions of users. Web browsers have become the *de facto* operating system for hosting these web-based applications (web apps). On the one hand,

the current web introduces a rich set of features enabling a variety of web apps, such as banking, shopping, social networking, etc. On the other hand, these features inevitably increase the complexity of web apps and browsers. Due to the complexity and outdated design of traditional browsers, attackers are able to carry out web-based attacks against web apps, browsers, and operating systems.

As explained in the introduction, the use of a browser kernel is highly advised. But, even when retrofitting a small such kernel to a browser, it still has to rely on the underlying operating system. In contrast, IBOS takes the modular design one step further to extend the principle to the operating system itself, introducing a small browser and operating system kernel that is the sole TCB of the whole system. The TCB makes it feasible to formally verify some of the key properties of web browsers, such as the same-origin policy.

The primary security policy that all modern browsers implement is the same-origin policy (SOP). The same-origin policy acts as a non-interference policy to ensure that web apps from different origins are isolated from each other. An origin is often defined as the <protocol, domain, port> tuple of the URL of a web app. Under the same-origin policy, a malicious web app from `attacker.com` should not be able to alter content and access sensitive information from `bank.com`. Unfortunately, due to their design, Chrome, Internet Explorer, Safari, and Firefox have to enforce the policy by using a number of checkers spread around the whole code base that consists of millions lines of code. Evidence shows that all of them have had trouble implementing the policy correctly [4].

3.2 IBOS architecture

IBOS proposes a highly modularized architecture of operating system and web browser by embracing principles from microkernel design. By exposing browser abstractions at the operating system kernel level, IBOS is able to remove all traditional OS and browser components and services from the TCB of the systems.

Figure 1 shows the architecture of IBOS. The IBOS architecture uses a thin kernel for managing hardware and facilitating message passing between processes. The system includes all traditional OS and browser components such as device drivers (e.g., networking interface card (NIC) driver), browser engines used for rendering web apps, and storage subsystem for storing cookies.

Some of the key goals of IBOS are the following, see [17] for all the goals and more detail:

- Security decisions happen at the lowest possible level: small TCB.
- Enough browser states and events exposed, so as to allow for security policy checking; this makes IBOS flexible to allow new browser security policies.

A key property of the IBOS browser is that *all communication*, i.e., all messages sent or received, *get transmitted through the* IBOS *kernel*. This is because the message passing is implemented as system calls, which of course go the the microkernel operating system, which is tightly integrated with the IBOS kernel.

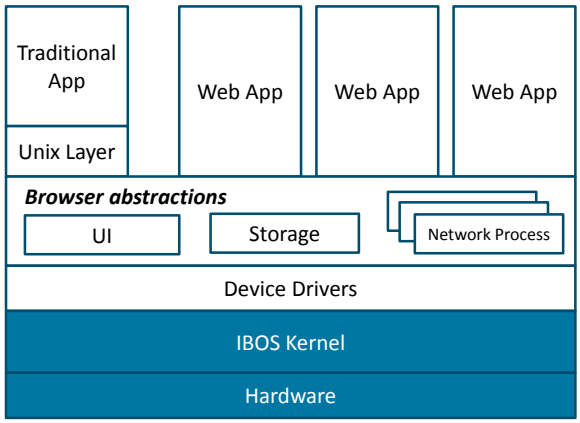

Fig. 1. IBOS Architecture

The components of the IBOS architecture which we want to highlight are the following three:

- **The IBOS kernel.** The IBOS kernel builds upon the L4Ka microkernel and is the central component of the IBOS web browser. It takes care of traditional OS tasks, e.g., process creation and application memory management. Message passing is based on the L4Ka::Pistachio message passing implementation, forcing all messages through the kernel, and specifically allows the checking of the security policies. Some of these policies are shown in Section 3.3
- **Network process.** The network process is responsible for HTTP requests. It transforms HTTP data into a TCP stream and in turn into a series of Ethernet frames which are passed to the NIC driver.
- **Web apps.** A new web app is created for each individual page visit of the user; specifically, whenever a link is clicked or a new URL is entered into the address bar. A web app sends out the HTTP request to the network process, parses HTML and runs JavaScript and renders web content to a tab. Each web app is labeled with the origin of the HTTP request used at creation.

3.3 Security goals

The modularized design and the small size of the TCB of IBOS enable the use of formal methods to verify the design's correctness. IBOS' use of small, simple, and exposed APIs allows us to model the system and reason about it. Using formal methods, we are able to check if the IBOS design preserves its security goals during an attack.

In IBOS, the goal is to minimize the TCB for web browsers and to simplify browser-based systems. To quantitatively evaluate its effort, the authors count the LOC in the IBOS TCB to be approximately 42K lines of code, which includes both the L4Ka kernel and integrated browser kernel. While the L4Ka kernel itself should also be formally verified to provide the security guarantee, we argue that it is a replaceable part in the TCB and there is already a verified kernel (seL4 [9]) in the L4 family that exposes a similar API that IBOS can use. As a result, in this paper we focus on proving the security properties based on the browser kernel design and implementation, and assume correct microkernel behavior.

Overall, we are going to verify the *same-origin policy* design in IBOS by verifying that the following invariants are upheld:

1. The kernel must route network requests from web page instances to the proper network process.
2. The kernel must route Ethernet frames from the network interface card (NIC) to the proper network process.
3. Ethernet frames from network processes to the NIC must have an IP address and TCP port that matches the origin of the network process.
4. HTTP data from network processes to web page instances must be from acceptable origins.
5. Network processes for different web page instances must remain isolated.
6. Isolation of the browser chrome (UI elements) and web page content displays.
7. Only the current tab can access the screen, mouse, and keyboard.
8. All components can only perform their designated functions.
9. The URL of the current tab is displayed to the user.

The same-origin policy is given by properties (1)–(7). Property (8) is another good property for IBOS, while property (9) aids in verifying property (7). Another important IBOS property we will verify is *address bar correctness*, that is, the address displayed in the address bar is always correct and cannot be spoofed.

3.4 Comparing the IBOS Approach to Commercial Browsers

IBOS enforces strong security guarantees by implementing a small kernel and exposing browser-related abstractions at the kernel level. By doing so, IBOS is able to ensure all critical browser-related messages pass through the kernel, which enforces security policies. Although commercial browsers, such as Chrome and ChromeOS [6], also use a browser kernel to validate messages, this browser kernel still runs on commodity operating systems and can only be as secure as

the underlying OS and system services. In contrast, IBOS has only a small kernel in its trusted computing base, which is what makes our modeling and verification effort feasible.

For performance, one concern is that the exclusive use of message passing in IBOS would cause a slowdown compared to traditional commercial browsers, such as Internet Explorer and Firefox. However, the authors show that the page loading speed of an unoptimized IBOS prototype is roughly equivalent to Chrome and Firefox for 6 popular web sites. Moreover, Chrome also uses message passing for most of the communication between its components, showing that using message passing in a browser implementation can be practical.

Although browser-based operating systems do limit the apps one can run on their system, we anticipate a system like IBOS being used in the same way users use ChromeOS where all user interface components are implemented using a browser.

4 Formal Specification of IBOS

Maude specifications were used systematically in the design of IBOS before its code was developed. Furthermore, we developed a more detailed formal specification of IBOS by doing a detailed study of the IBOS C++ source code to reflect all important details. This detailed specification was then subjected to thorough review by a joint team of modelers and developers to ensure that it faithfully reflected the implementation, and, in one case, to detect an implementation flaw. In this way, the formal specification has been further refined during this phase. Thus, we made sure that the original design intentions, the source code, and the formal specification matched correctly.

For the full formal model with detailed explanations see the PhD thesis [15]. In this section we point out key properties and give a general flavor of the model. At the top level, our state space is made up of objects with an object identifier, a type, and a set of attributes. Each network process, web app, and the kernel is modeled as a single object. Each of these components runs in parallel and is independent, except for communication. We show the form of the distributed state of the model in Figure 2. In that figure all objects outside the kernel are shown as rectangles. Note that pipes are a special kind of object that connects the objects at its left and right end. Other than that, arrows show connectivity. The ellipses inside the kernel contain relevant pieces of the kernel, that are not objects themselves, i.e., they are not independent components. There will of course be multiple instances of objects for most classes, except for the NIC, display and web app manager.

Let us start by looking at the kernel in more detail, particularly at the message passing mechanism. First, we present more information on the messages. All messages are passed as system calls of the underlying micro kernel operating system, where the browser-specific part of the message is encapsulated in the system call. The message part specific to the browser has the following format, which we call the `payload` of the encapsulating system call:

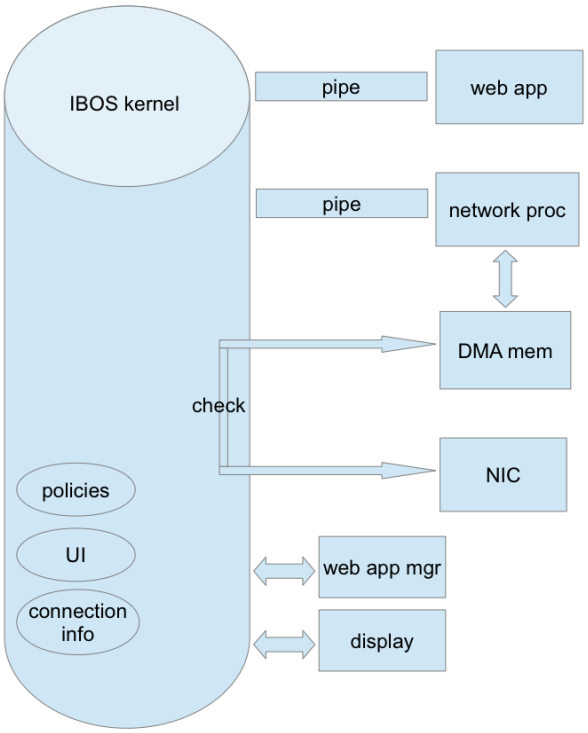

Fig. 2. IBOS Model State

```
op payload : Oid Oid MsgType String  -> Payload [ctor] .
```

The arguments of payload are the sender (as `Oid`), the receiver (as `Oid`), the message type (as `MsgType`), and an argument commonly containing the URL that is requested or sent (as `String`). We have simplified a little here and left out some extra arguments that are in the model, but which are not relevant for the present purposes. The sort `Oid` is that of object or process identifiers. Each web app, network process, etc., has a unique `Oid`. Note that the correct sender `Oid` is enforced by the kernel, as it knows which process sent the system call encapsulating this `payload`.

The actual message is then built using the `payload` and system call type:

```
op msg : SyscallType Payload -> Message [ctor] .
op OPOS-SYSCALL-FD-SEND-MESSAGE : -> SyscallType .
```

where `OPOS-SYSCALL-FD-SEND-MESSAGE` is the most commonly used type of system call for sending browser messages.

To model the fact that the kernel knows which process actually sent a message (as a system call) and to make sure that in the model no two processes can send messages directly to each other, but are forced to send messages via the

kernel, the model defines one pipe object per process (using the same `Oid` as the associated process), which contains two one-way pipes, going to the kernel from the process and going to the process from the kernel:

```
op pipe : -> Cid [ctor] .

op fromKernel : MessageList -> Attribute [ctor] .
op toKernel : MessageList -> Attribute [ctor] .
```

Let us show an example pipe object for the process with `1050` as `Oid` with two buffers which currently holds no message (`mt`) going either way:

```
< 1050 : pipe | fromKernel(mt), toKernel(mt) >
```

Suppose this process wants to send, for example, the message:

```
msg(OPOS-SYSCALL-FD-SEND-MESSAGE, payload(1050, 256,
     MSG-FETCH-URL, l(http,dom("test"),port(81))))
```

This message comes from web app `1050` and goes to network process `256`, sending the message to fetch a URL (`MSG-FETCH-URL`) from the (fictional) domain `http://test:81`. This message would then be appended to the list of messages held in `toKernel` in the pipe object. The kernel enforces correct sender `Oid` based on the pipe's id by simply changing the given sender `Oid`, if necessary.

As part of the policy checking when a network process and a web app communicate, their connection is checked. This means that both of them need to be linked to the same origin. This is modeled by the equation:

```
eq < kernel-id : kernel |
     handledCurrently(checkConnection(Num:Nat, Num':Nat, M)) ,
     weblabels(pi(Num':Nat, L:Label), WPIS:WebappProcInfoSet) ,
     networklabels(pi(Num:Nat, L:Label, L':Label),
                   NPIS:NetworkProcInfoSet) , Att >
 =  < kernel-id : kernel |
      handledCurrently(M) ,
      weblabels(pi(Num':Nat, L:Label), WPIS:WebappProcInfoSet) ,
      networklabels(pi(Num:Nat, L:Label, L':Label),
      NPIS:NetworkProcInfoSet) , Att > .
```

The property being checked here is that the receiving web app with id `Num':Nat` is associated to a URL `L:Label` in the kernel storage for web app connections `weblabels`, and that the sending network process with id `Num:Nat` is associated with the same URL `L:Label` in the network process connection storage `networklabels`. Then the message is simply being passed on, by dropping the `checkConnection` wrapper around the message `M`. The kernel is only handling one thing at a time, which is stored in `handledCurrently`. Once the current instruction has been dealt with, any of the currently incoming messages can become the next message to be executed. This is modeled by the rule:

```
rl [kernelReceivesOPMessage] :
```

```
   < kernel-id : kernel |
       handledCurrently(mt) , msgPolicy(MP), Att >
   < ID : pipe | toKernel(msg(ST:SyscallType,
        payload(N, N', M:MsgType, S:String)), ML) , Att2 >
=> < kernel-id : kernel |
      handledCurrently(policyAllows(msg(ST:SyscallType,
        payload(ID, N', M:MsgType, S:String)), MP)) ,
      msgPolicy(MP), Att >
  < ID : pipe | toKernel(ML) , Att2 > .
```

Note that the kernel does not take the message to be dealt with directly, but wraps the actual message inside the `policyAllows` operator, together with the set of message policies `MP` as an extra argument, which is an attribute of the kernel wrapped in `msgPolicy`. Also, in the message the sender id `N`, which was given by the sender, is forcibly changed to the actual sender id `ID`, which is the process id of the pipe (and thus the associated process).

For the network process we are using (as does IBOS) the process ids 256 through 1023. The attributes of a network process are:

```
op returnTo : ProcId -> Attribute [ctor] .
op in : LabelList -> Attribute [ctor] .
op out : LabelList -> Attribute [ctor] .
```

The `returnTo` attribute stores the process id of the web app that this network process will return data to, while the attributes `in` and `out` hold the lists of labels (representing URLs) that the network process will ask data from and has received data from already. We simplify here by using a URL to represent its data, instead of using its actual HTML code.

For web apps we are using the process ids 1024 through 1055 with attributes:

```
op rendered : Label -> Attribute [ctor] .
op URL : Label -> Attribute [ctor] .
op loading : Nat -> Attribute [ctor] .
```

The label inside `rendered` is the URL for which the web app has put the data on the screen, provided it is the active web app. The label inside `URL` is the location where this web app wants to load data from. `loading` is just a binary flag indicating whether the web app has already sent a request to load data. Initially, the `rendered` field for a new web app will be empty, and `loading` is `0`, meaning that it has not yet started to load. The following equation sends the message to start loading:

```
eq < N : proc | rendered(L) , URL(L') , loading(0) , Att >
   < N : pipe | toKernel(ML) , Att2 >
 = < N : proc | rendered(L) , URL(L') , loading(1) , Att >
   < N : pipe | toKernel(ML, msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
        payload(N, network-id, MSG-FETCH-URL, L'))) , Att2 > .
```

The message is sent to fetch the data from URL `L'` and the `loading` attribute changes to `1`. On return of the requested data, `rendered` will change to `L'`.

The hardware pieces of Figure 1, video card, NIC, etc., are not modeled in any detail. Only the NIC is modeled, and it receives target URLs from the memory set aside for this purpose through the kernel, and then, after a potential delay, returns the representation of the resulting data.

This model uncovered an issue with the display memory, which turned out to be a bug. It was found in the model and could then be fixed in both model and implementation. The issue was that when switching web apps, the newly active web app would sometimes not get access to the actual display memory, which would then simply stay blank. The model let us figure out how and why this happened. This is not a security concern, but actually a usability issue, as the tab in question becomes useless, but cannot be abused for malicious purposes.

5 Formal Verification of IBOS Security

The verification of IBOS design security is based on the formal model explained in the prior section. We simply assume that the underlying microkernel operating system performs its functions correctly. Of course, in order to not have to rely on this we could instead use seL4 [9], which has been verified.

An important property for a web browser is the trustworthiness of user interface elements. This is crucial to counter spoofing attacks. Particularly, the address bar needs to be trustworthy, so that the user always knows which site is currently being visited. It is truly important to know whether the currently visited site is really his/her banking web site, where entering credentials is fine, or if it is instead a phishing web site, where if the user enters his/her account information monetary loss is imminent. We all know that it is possible, even simple, for malicious attackers to create phishing web sites that are indistinguishable on the surface from the real web sites. A careful user should be able to trust the address bar, to prevent such phishing from succeeding. Also see [3,15] about the address bar spoofing possibilities we found in an analysis of Internet Explorer.

Similarly important is the correct implementation of security policies in the browser. The same-origin policy, presented in Section 3.3, is one such policy that assures the user that his private information, say from a banking web site, will not be leaked to another web site, with possibly malicious intentions.

As IBOS has been designed with security in mind, our goal in this section is not just to find possible flaws that could be abused by attackers. Our goal actually is to be able to prove that no such address bar spoofing attacks are possible, as well as to verify the correct implementation of SOP. First, let us show the operator which drives the search, simulating user input, `inspect-space`:

```
op inspect-space :  -> Configuration .
eq inspect-space = < testMsg : testMsg | cmd( inspect ) > .
```

where `testMsg` is a wrapping process, which allows this to be put at the top level of our multi-set of processes, and `cmd` is a wrapper allowing this to follow the usual way of storing information in process attributes. The key here is the `inspect` command. We will call the rules for `inspect` the *trigger rules*, and write

them as R_T. All other rules belong to the *internal rules* of the model, written as R_I. We are working modulo the equations E. So we are actually rewriting with $\rightarrow_{R_{(I \cup T)/E}}$, which can be split into $\rightarrow_{R_{I/E}}$ and $\rightarrow_{R_{T/E}}$. We will use the shorthands $\rightarrow_I$ and $\rightarrow_{I/E}$ (resp. $\rightarrow_T$ and $\rightarrow_{T/E}$) to represent $\rightarrow_{R_{I/E}}$ (resp. $\rightarrow_{R_{T/E}}$).

```
op inspect : -> Cmd .
op inspect : Nat -> Cmd .
rl inspect => inspect(3) .
rl inspect(0) => mtCmdList .
rl inspect(s(N:Nat)) => new-url , inspect(N:Nat) .
rl inspect(s(N:Nat)) => switch-tab , inspect(N:Nat) .
```

This shows that `inspect` is unrolled step by step. The number `3` can of course be changed, but that number is picked in particular so that two web apps can be created and the tab can then be switched as well. At each step either a `switch-tab` or `new-url` will be generated. This simulates user input. As `inspect` is defined by rules, the `search` command will create all possible combinations. We omit how `new-url` gets assigned a new URL and how `switch-tab` picks any of the web apps to be the new active web app.

Internal Normalization Between Trigger Rules. We observe that there is no interference between internal rules I and trigger rules T, i.e., we can re-order them in any way we please. In particular, we like to normalize with the internal rules after each execution of a trigger rule. That means, for execution using both internal rules and trigger rules, $\rightarrow^*_{(T \cup I)}$, we will rearrange that to $\rightarrow_T \rightarrow^!_I \ldots \rightarrow_T \rightarrow^!_I \ldots \rightarrow_T \rightarrow^*_I$, where $\rightarrow^!_T$ denotes a terminating subsequence. The last set of internal rules does not have to be carried all the way to normalization, to take into account the fact that the combination of trigger and internal rules might not normalize either. Let us state this formally as a lemma, noting that by $\rightarrow^i_{T/E}$ we mean the i-th use of a rule from T/E:

Lemma 1. *Given terms s_1 and s_2, for any chain of rewrites of the form $s_1 \rightarrow^*_{(T \cup I)/E} s_2$, with n uses of trigger rule, we can rearrange that sequence, using the same rewrites, to $s_1 \rightarrow^1_{T/E} \rightarrow^!_{I/E} \ldots \rightarrow^i_{T/E} \rightarrow^!_{I/E} \ldots \rightarrow^n_{T/E} \rightarrow^*_{I/E} s_2$.*

This lemma is based on the minimal overlap of the trigger rules in T and the internal rules in I. In particular, no trigger rule step is ever influenced by any internal rule step, that is, trigger rule steps can either be taken, or not taken, independently of anything happening with the internal rules. Conversely, a trigger rule step may enable additional internal rule steps to be taken, but does not disable internal rule steps that have been available already. The entire proof for this is included in [15]. We can now consider the effect of each trigger rule on the state by itself. We let the model do all internal computations until finished before using another trigger rule step.

5.1 Address Bar Correctness Verification

Address bar correctness in the model means that the content of the displayed page is always from the address which is displayed in the address bar. In our

model, the kernel keeps track of the address bar by means of the data stored in the `displayedTopBar`. The source of the content being displayed is stored in the display process abstraction, which has the `displayedContent` field to store the information. The content of both these fields needs to be the same at all times. Only when there currently is no content in one of the two field, which is modeled by the `about-blank` URL, the other one can have any value.

To motivate the property of address bar correctness, note that the address bar, and the content as stored in the display process, are both stateless objects. They have no memory, but only know what is stored in them right now.

Both the address bar and the display content are only changed due to the current web app interacting with the kernel when created or when the tab is switched to it. To create a mis-match between the two, two different URLs are all that is needed, which can be provided by just two web apps. This allows us to make the reduction that only the last two web apps that are on the screen need to be taken into account. The rest of the browser model state and the length of the run of the browser model is irrelevant and thus can be abstracted away.

Assume we needed to consider a third web app, then that would only be the case if that web app made a change to either of the two objects in question; but then one of the other two does not make a change (or does a duplicate one), so then that other web app becomes irrelevant and we are back to the case of two web apps. If there was a way for more than two web apps to create such a mis-match, then the deciding last step (we would stop at such a mis-matching point) must be either a new web app being added or the tab being switched. But then, that whole trace of actions and number of web apps can be simplified to just the state before that last action, with only the old active web app and the new active web app taken into account to create the exact same mis-match. Now we can focus on the interaction of only two web apps, which requires search up to depth three, due to the need of also allowing a tab switch.

We now present our theorem for the address bar correctness.

Theorem 1. *The property of address bar correctness holds for any rewrite sequence, using any number of trigger rule steps.*

For the detailed proof, see [15]. We will show that bounded model checking analysis of all sequences with at most 3 trigger rules finds no possible violation. So, the address bar is correct for all sequences with at most 3 trigger rules. A reduction from longer sequences to sequences of at most length 3 then proves the theorem. This means that the correctness extends to sequences with *any* number of trigger rules being used. Let us start with that lemma, which is proved by a detailed case analysis using Lemma 1, see [15]

Lemma 2 (Reduction). *Any sequence of trigger rule steps that leads to a violation of the address bar correctness and uses 4 or more trigger rule steps can be reduced by a step. This yields that all possible trigger rule sequences leading to a violation must be of length 3 or less.*

Now that we have the reduction to 3 trigger rule steps, we can use bounded model-checking to analyze this finite state space. We start the model-checking

search for potential attacks, in the form of a mismatch of these two fields, from an initialized kernel, together with the driver `inspect-space`. We are looking for any configuration in which there is a mismatch between the value of `displayedTopBar` and `displayedContent`. If no solution to this search is found, then there is no attack for this bounded case.

```
search init-simp-kernel
inspect-space =>* X:Configuration
  < kernel-id : kernel | Att:AttributeSet ,
          displayedTopBar(URL:Label) >
  < display-id : proc |
    displayedContent(URL':Label), Att2:AttributeSet >
such that URL:Label =/= URL':Label
     and  URL:Label =/= about-blank
     and  URL':Label =/= about-blank .
```

Indeed, when we run this search command we find no solutions as result:

```
No solution.
states: 247743  rewrites: 3663864 in 247886ms cpu
               (248055ms real) (14780 rewrites/second)
```

Together with the reduction lemma, Lemma 2, this completes the proof.

Bounded model-checking is indeed required here, as the active web app can change the content on screen after the address bar has been set. Of course, that web app is associated to the URL in the address bar in the kernel, so the kernel will not allow the web app to access any other origins and thus the address bar correctness will hold.

5.2 Same-Origin Policy Verification

After the proof of the address bar correctness we now prove that IBOS implements the same-origin policy correctly, that is, that it satisfies properties (1)–(7) in Section 3.3. We will now look more closely at those security requirements which result in the browser implementing SOP.

To analyze the SOP property of our browser, we use the model of the internal logic of the browser we have introduced; it includes the policies being enforced by the kernel. We already noted that all messages go through the kernel and thus are subject to being checked with respect to the policies. We then also have to create canonical messages that different components can try to send to each other. That is, we need a small set of messages that is *generic*, so that the instances of these generic messages can cover all messages. Then the analysis can in fact verify that none of those messages can reach disallowed destinations. We again use a reduction to a limited number of required trigger steps and then use model checking to show the property for this smaller state space.

See Section 3.3 for the whole list of SOP properties. Let us consider the first property of SOP in more detail:

– (1) *The kernel must route network requests from web page instances to the proper network process.*

Let us first motivate at a high level why this should be true before we state the lemma and show the model-checking analysis afterwards. Simply said, each web page instance and each network process have an associated URL which identify them to the kernel, in addition to their actual process id. This URL is the URL they are allowed to communicate with. Now, whenever a web page instance tries to communicate with a network process, the kernel checks the process id and associated URL for both. For this purpose, the kernel stores a mapping of process id to URLs. If no appropriate network process exists, a new one will be created by the kernel. In practice, the kernel (and its modeling) enforces that only matching processes communicate. For checking property (1) we look at each message that is received by any network process and compare the URLs of sender and receiver using the kernel's mapping. Note that sender and receiver names cannot be forged as these are their process ids and enforced by the kernel based on the underlying guarantees of the operating system.

Indeed, the execution for property (1) does not make use of a history of what happened before, but only of the current assignment of each process to URL. We can abstract away from a long sequence of network requests to simply one single network request. As the state is generic and the correctness of the property only depends on one network request, if we can show the absence of errors for this one network request, we know that any arbitrary number of them still will not exhibit any errors. Otherwise, we could take just that network request which triggers the error and use it to get the error by itself, contradicting the fact that we show that no single message creates an error.

Checking property (1) then boils down to checking executions (up to some depth of input), from canonical starting points, to see whether there is a mis-match between URLs in the resulting configuration for any message. If there is no mis-match for all starting points, then all communications have been legal and property (1) is actually proved. We can limit the depth of execution, i.e., the number of messages being considered, and still be complete. Each message is generic and representative of a set of messages. The reason we can limit the depth is that if the property would turn out to be possibly violated at an arbitrary number of messages, then that final message triggering the failure will only have one source process and one destination process. That violation can then be boiled down to the triggering network request, and the setup for those involved two processes, which would be a total depth of three actions.

Now we can state the theorem for SOP, whose detailed proof is given in [15]:

Theorem 2. *The Same-Origin Policy holds for any rewrite sequence, using any number of trigger rule steps.*

The proof of this theorem consists of the proofs for all of the SOP properties (1)–(7). To illustrate this we give the lemma for property (1).

Lemma 3. *The property* (1) *holds for any rewrite sequence, using any number of trigger rule steps.*

The proof we give is based on the number of trigger rule steps needed to find a violation. We have used bounded model-checking to show there is no violation

up to the bound. Assume there is a violation, then we pick one of the sequences that lead to such a violation with the smallest number of trigger rule steps. That number must be bigger than the bound. We then analyze that sequence and we find at least one step that is not needed. By that we mean that after removing this one trigger rule step the same violation is still reached, but now the sequence is one trigger rule step shorter. This contradicts that we pick the sequence with the smallest number of trigger rule steps and thus there is no violation for any number of trigger rule steps. We can state this as a reduction lemma again:

Lemma 4 (Reduction). *Any sequence of trigger rule steps that leads to a violation of property* (1) *and uses 4 or more trigger rule steps can be reduced by a step. Thus, all possible trigger rule sequences leading to a violation of property* (1) *must be of length 3 or less.*

The proof in [15, Chapter 4] explains this reduction by which we extend the bounded model-checking proof for sequences with at most three triggers to sequences with *any* number of triggers in detail.

The following search returns no solution, meaning that no illegal (according to SOP) communication happened.

```
search init-simp-kernel inspect-space =>*
X:Configuration < N:Nat : pipe |  toKernel(ML:MessageList) ,
   fromKernel(msg(OPOS-SYSCALL-FD-SEND-MESSAGE,
         payload(Num:Nat, N:Nat, MSG-FETCH-URL, L1:Label)),
         ML':MessageList) , Att:AttributeSet >
< kernel-id : kernel | Att2:AttributeSet ,
     weblabels(pi(Num:Nat,L1':Label), WAPIS:WebappProcInfoSet) ,
     networklabels(pi(N:Nat, L2':Label, L2:Label),
                   NPIS:NetworkProcInfoSet)  ,
     displayedTopBar(URL:Label) >
such that  L1:Label =/= L2:Label or L1':Label =/= L2':Label .
```

As the above bounded model-checking did not return any violations, no illegal messages were passed. All network requests indeed end up going to the proper network processes. This bounded model-checking analysis proves the base cases of up to 3 trigger steps, while together with the reduction lemma, Lemma 4, this yields the proof for all possible sequences.

The model contains about 20 rules and 60 equations and the bounded model-checking takes between 10 and 20 minutes to check each property.

For all of the remaining properties (2)–(9) we give similar proofs in [15]. We omit their statement and explanation due to space reasons in this paper. Altogether, this shows that the SOP is correctly implemented for the IBOS formal specification, and thus for the IBOS design.

6 Related Work and Conclusions

Let us consider some previous work formally verifying the design or implementation of operating system kernel and browsers. In seL4 [9], the authors use

a self-developed framework to verify the correctness of a L4 type microkernel, which provides the foundation of further proof of systems built on top it, such as our work in this paper. Heiser *et al.* discuss some possibilities in system security once one has a truly trustworthy kernel [8]. One of those possibilities is providing even stronger security guarantees for a web browser, like we do in this paper. On the browser side, the designers of the OP browser [7] use formal methods to verify some security properties of their design, in particular, they are concerned with whether or not the security indicators behave as expected. Formal methods have also been used to check properties of the status bar and address bar in Internet Explorer [3]. Akhawe *et al.* also propose an abstract formal model of web security and use it to analyze the security of several web apps [1]. However, even though those approaches improve the security of the corresponding systems, these systems still rely on a large TCB that could not be verified. In our work we were able to obtain an executable specification of the IBOS design thanks to the substantially smaller TCB of the web browser. This was key to prove security properties of the IBOS design.

In this paper we present a correct-by-construction browser with analysis of its correctness at the design level. For this, we show how useful a formal specification of such a modular piece of software is. It allowed us to first find bugs, to fix them, and then to prove correctness of the address bar and the same-origin policy.

There are a few follow up projects that would be useful as future work. First, now that a design that has been analyzed in detail exists, it would be highly desirable to analyze the implementation using (semi-)automatic source code verification tools such as matching logic [14]. Another way of further increasing the confidence in IBOS would be to actually use the proven secure microkernel seL4 or to develop a new proof of security for IBOS' underlying microkernel. In the other direction, it would also be interesting to consider generating source code from the model directly, i.e., code synthesis.

Acknowledgments This research was partially supported by NSF Grant CCF 09-05584 and AFOSR Grant FA8750-11-2-0084 as well as by grant N0014-09-1-0743 from the Office of Naval Research, AFOSR MURI grant FA9550-09-01-0539, NSF grant CNS 0831212, and by Intel through the ISTC for Secure Computing.

References

1. D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 290–304, Washington, DC, USA, 2010. IEEE Computer Society.
2. A. Barth, C. Jackson, C. Reis, and The Google Chrome Team. The security architecture of the chromium browser, 2008. `http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf`.
3. S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy*, pages 71–85. IEEE Computer Society, 2007.
4. S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In P. Ning, S. D. C.

di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 2–11. ACM, 2007.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
6. Google Inc. Chromium OS, 2010. `http://www.chromium.org/chromium-os`.
7. C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, May 2008.
8. G. Heiser, L. Ryzhyk, M. Von Tessin, and A. Budzynowski. What if you could actually trust your kernel? In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 27–27, Berkeley, CA, USA, 2011. USENIX Association.
9. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
11. A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*, February 2006.
12. N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proceedings of the 17th Usenix Security Symposium*, pages 1–15, July 2008.
13. N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of Web-based malware. In *Proceedings of the 2007 Workshop on Hot Topics in Understanding Botnets (HotBots)*, April 2007.
14. G. Rosu and A. Stefanescu. Matching logic: a new program verification approach. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 868–871. ACM, 2011.
15. R. Sasse. *Security Models in Rewriting Logic for Cryptographic Protocols and Browsers*. PhD thesis, University of Illinois at Urbana-Champaign, July 2012. Current Draft available at `http://formal.cs.illinois.edu/rsasse/dissertation-code/thesis-draft.pdf`.
16. Symantec. Symantec internet security threat report, 2011. `http://www.symantec.com/business/threatreport`.
17. S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In R. H. Arpaci-Dusseau and B. Chen, editors, *OSDI*, pages 17–32. USENIX Association, 2010.
18. TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, 2003.
19. H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 2009 USENIX Security Symposium*, August 2009.
20. Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web sites that exploit browser vulnerabilities. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*, February 2006.