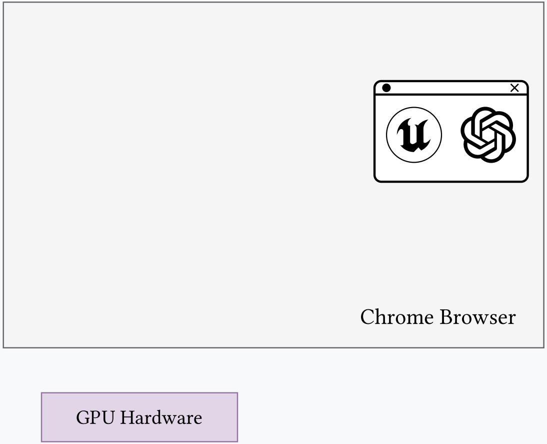
DarthShader:
Fuzzing WebGPU Shader Translators & Compilers

Lukas Bernhard, Nico Schiller,
Moritz Schloegel, Nils Bars, Thorsten Holz

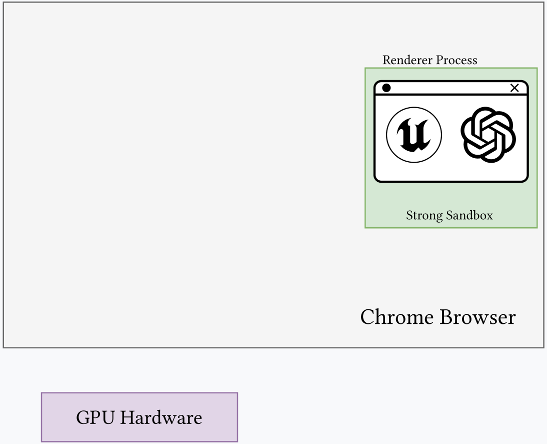
October 14-18, 2024, CCS '24, Salt Lake City, U.S.A.

CISPA

WebGPU - Exposing GPUs to the Web



WebGPU - Exposing GPUs to the Web



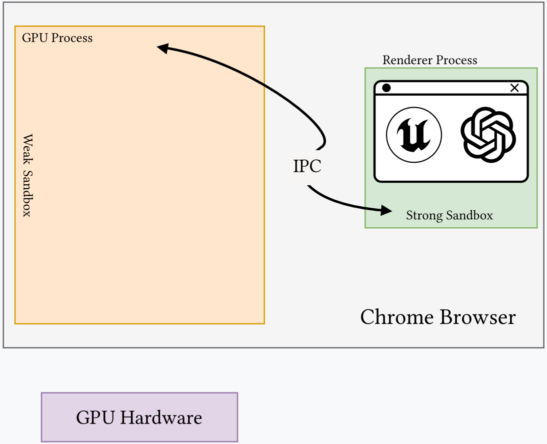
Renderer Process

Strong Sandbox

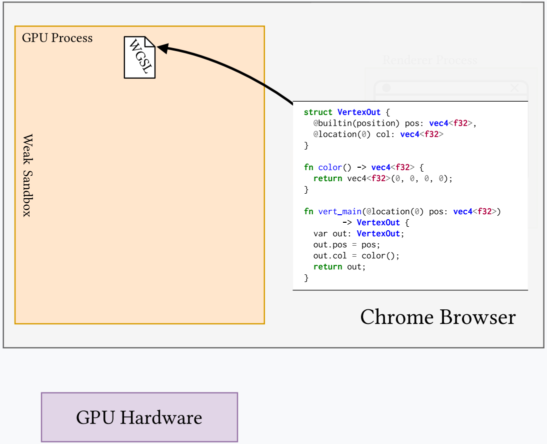
Chrome Browser

GPU Hardware

WebGPU - Exposing GPUs to the Web

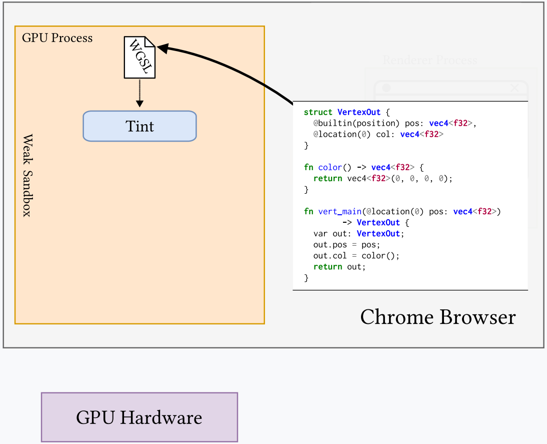


WebGPU Shading Language - WGSL



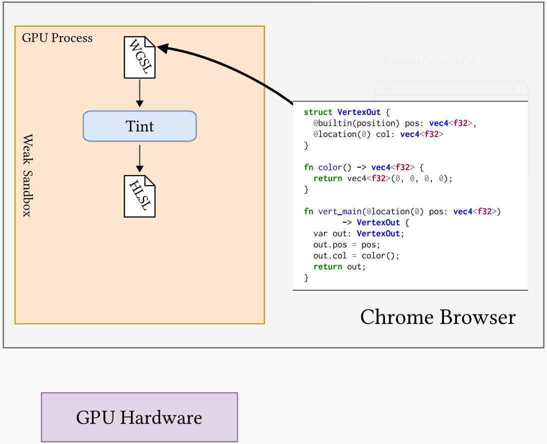
```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

WGSL - Compilation Pipeline



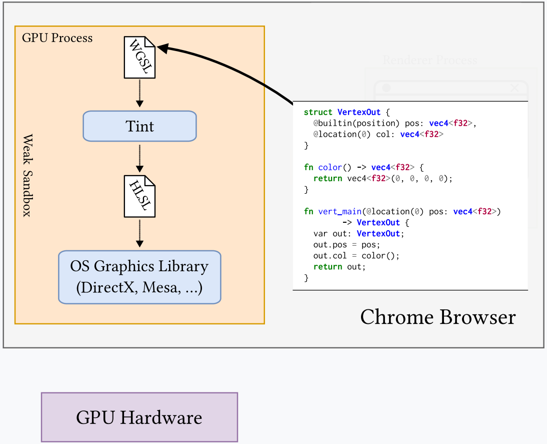
```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

WGSL - Compilation Pipeline



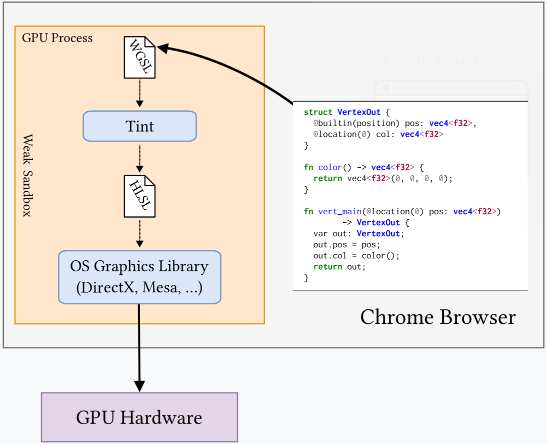
```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

WGSL - WebGPU Shading Language



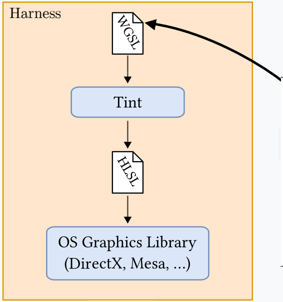
```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

WGSL - WebGPU Shading Language



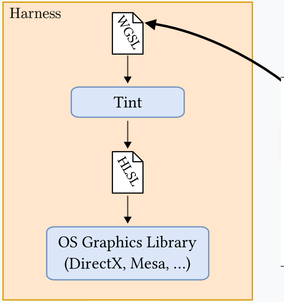
```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

Fuzzing - Harness



```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

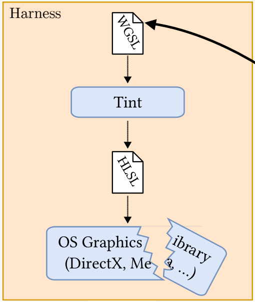
Fuzzing - IR Mutations



```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

```
Expressions:                    // Types:
[1]: FunctionArgument(0)        // vec4<f32>
[2]: LocalVariable([1])         // VertexOut
[3]: Access { [2], idx: 0 }     // vec4<f32>
[4]: Access { [2], idx: 1 }     // vec4<f32>
[5]: CallResult([1])            // vec4<f32>
[6]: Load { ptr: [2] }}         // VertexOut

Statements:
EmitExpr([3])
Store { pointer: [3], value: [1] }
EmitExpr([4])
Call { fun: color, args: [], res: [5] }
Store { pointer: [4], value: [5] }
EmitExpr([6])
Return { value: Some([6]) }
```

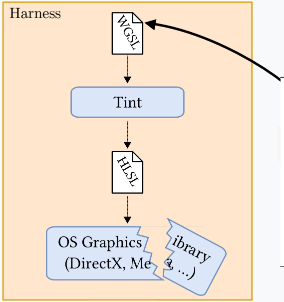
Fuzzing - IR Mutations



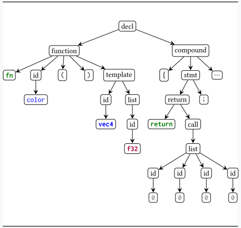
```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

```
Expressions:                    // Types:
[1]: FunctionArgument(0)        // vec4<f32>
[2]: LocalVariable([1])         // VertexOut
[3]: Access { [2], idx: 0 } // vec4<f32>
[4]: Access { [2], idx: 1 } // vec4<f32>
[5]: CallResult([1])            // vec4<f32>
[6]: Load { ptr: [2] }}         // VertexOut

Statements:
EmitExpr([3])
Store { pointer: [3], value: [1] }
EmitExpr([4])
Call { fun: color, args: [], res: [5] }
Store { pointer: [4], value: [5] }
EmitExpr([6])
Return { value: Some([6]) }
```

Fuzzing - AST Mutations

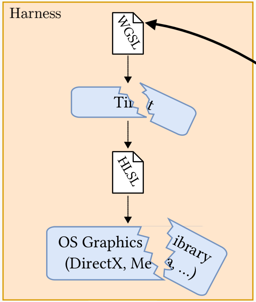


```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

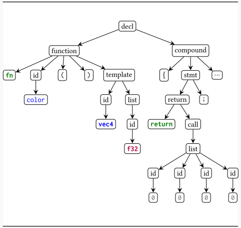


```
Expressions:                    // Types:
[1]: FunctionArgument(0)        // vec4<f32>
[2]: LocalVariable([1])         // VertexOut
[3]: Access { [2], idx: 0 } // vec4<f32>
[4]: Access { [2], idx: 1 } // vec4<f32>
[5]: CallResult([1])            // vec4<f32>
[6]: Load { ptr: [2] }}         // VertexOut

Statements:
EmitExpr([3])
Store { pointer: [3], value: [1] }
EmitExpr([4])
Call { fun: color, args: [], res: [5] }
Store { pointer: [4], value: [5] }
EmitExpr([6])
Return { value: Some([6]) }
```

Fuzzing - AST Mutations

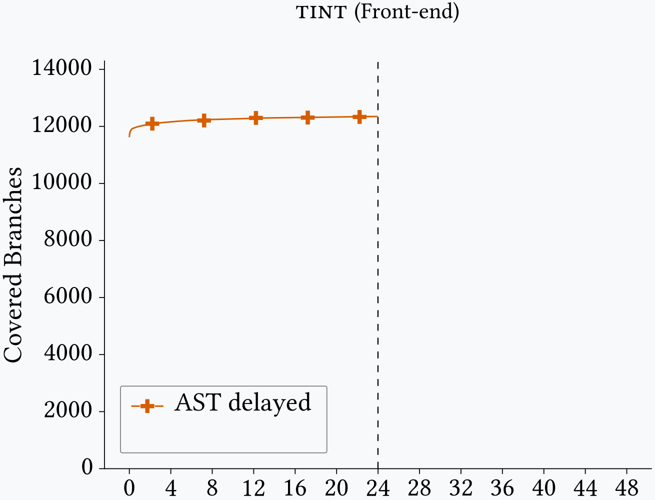


```
struct VertexOut {
  @builtin(position) pos: vec4<f32>,
  @location(0) col: vec4<f32>
}

fn color() -> vec4<f32> {
  return vec4<f32>(0, 0, 0, 0);
}

fn vert_main(@location(0) pos: vec4<f32>)
        -> VertexOut {
  var out: VertexOut;
  out.pos = pos;
  out.col = color();
  return out;
}
```

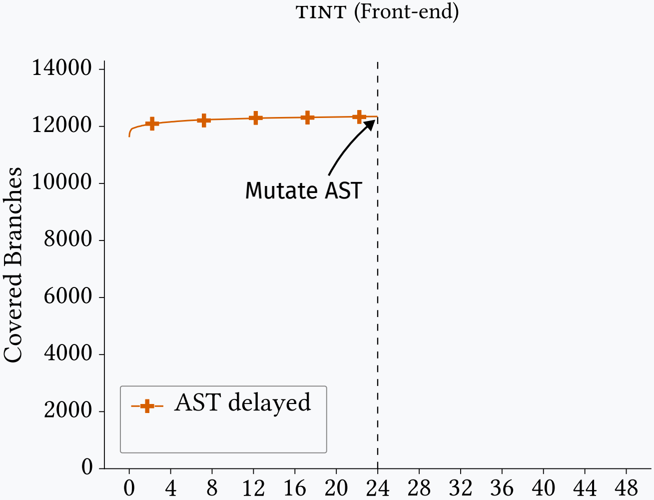


```
Expressions:                  // Types:
[1]: FunctionArgument(0)      // vec4<f32>
[2]: LocalVariable([1])       // VertexOut
[3]: Access { [2], idx: 0 } // vec4<f32>
[4]: Access { [2], idx: 1 } // vec4<f32>
[5]: CallResult([1])          // vec4<f32>
[6]: Load { ptr: [2] }}       // VertexOut

Statements:
EmitExpr([3])
Store { pointer: [3], value: [1] }
EmitExpr([4])
Call { fun: color, args: [], res: [5] }
Store { pointer: [4], value: [5] }
EmitExpr([6])
Return { value: Some([6]) }
```

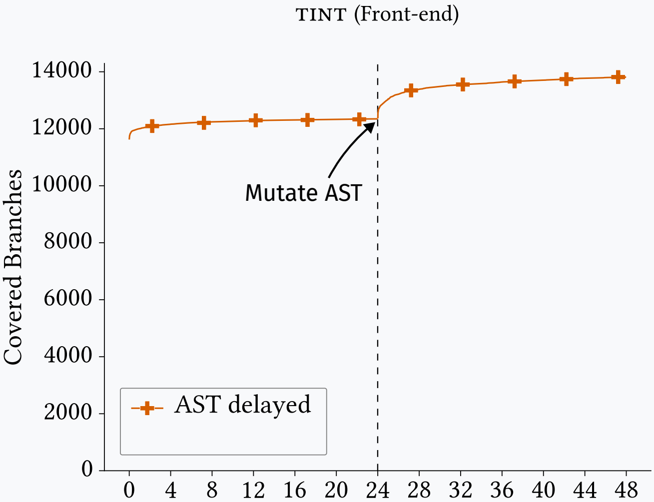
Design Verification



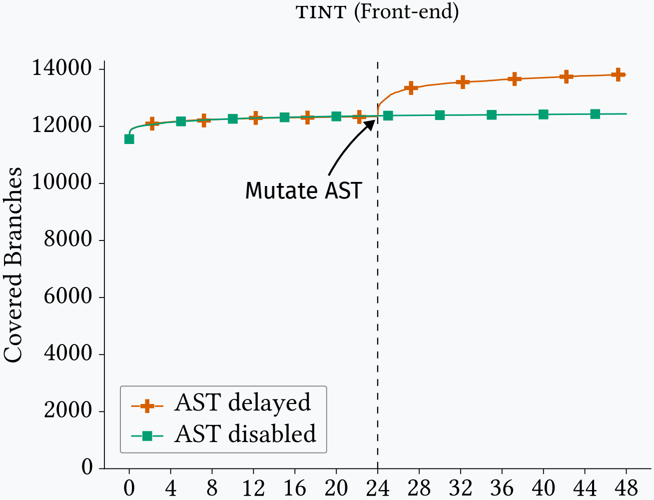
Design Verification



Design Verification



Design Verification



Bugs

SUT	Bug ID	Browser	Status
angle	chromium 329271490	◉ ⟲ 🧭	fixed
dxcompiler	chromium 1513069	◉	open
dxcompiler	CVE-2024-2885	◉	fixed
dxcompiler	CVE-2024-3515	◉	fixed
dxcompiler	CVE-2024-4948	◉	fixed
dxcompiler	CVE-2024-4060	◉	fixed
dxcompiler	CVE-2024-4368	◉	fixed
dxcompiler	CVE-2024-5160	◉	fixed
dxcompiler	CVE-2024-5494	◉	fixed
tint	tint 2190	◉	fixed
tint	tint 2201	◉	fixed
tint	tint 2202	◉	fixed
tint	tint 2055	◉	fixed
tint	tint 2056	◉	fixed
tint	tint 2058	◉	fixed
tint	tint 2068	◉	fixed
tint	tint 2076	◉	fixed
tint	tint 2077	◉	fixed
tint	tint 2078	◉	fixed
tint	tint 2079	◉	fixed

Bugs, Bugs Everywhere

SUT	Bug ID	Browser	Status
angle	chromium 329271490	◎ 🦊 🧭	fixed
dxcompiler	chromium 1513069	◎	open
dxcompiler	CVE-2024-2885	◎	fixed
dxcompiler	CVE-2024-3515	◎	fixed
dxcompiler	CVE-2024-4948	◎	fixed
dxcompiler	CVE-2024-4060	◎	fixed
dxcompiler	CVE-2024-4368	◎	fixed
dxcompiler	CVE-2024-5160	◎	fixed
dxcompiler	CVE-2024-5494	◎	fixed
tint	tint 2190	◎	fixed
tint	tint 2201	◎	fixed
tint	tint 2202	◎	fixed
tint	tint 2055	◎	fixed
tint	tint 2056	◎	fixed
tint	tint 2058	◎	fixed
tint	tint 2068	◎	fixed
tint	tint 2076	◎	fixed
tint	tint 2077	◎	fixed
tint	tint 2078	◎	fixed
tint	tint 2079	◎	fixed

SUT	Bug ID	Browser	Status
dxcompiler	CVE-2024-5495	◎	fixed
dxcompiler	CVE-2024-6102	◎	fixed
dxcompiler	CVE-2024-5831	◎	fixed
dxcompiler	CVE-2024-5832	◎	fixed
dxcompiler	CVE-2024-6290	◎	fixed
dxcompiler	CVE-2024-6292	◎	fixed
dxcompiler	CVE-2024-6103	◎	fixed
dxcompiler	CVE-2024-6293	◎	fixed
dxcompiler	CVE-2024-6991	◎	fixed
tint	tint 2092	◎	open
tint	tint 2194	◎	open
naga	naga 2560	🦊	fixed
naga	naga 2568	🦊	fixed
naga	wgpu 4547	🦊	open
naga	wgpu 4512	🦊	open
naga	wgpu 4513	🦊	open
naga	wgpu 5547	🦊	fixed
wgslc	webkit 268148	🧭	open
wgslc	webkit 273407	🧭	fixed
wgslc	webkit 273411	🧭	fixed

Takeaways

- GPU stack is exposed by WebGPU

DarthShader: Fuzzing WebGPU Shader Translators & Compilers

Lukas Bernhard
CISPA Helmholtz Center for
Information Security
Germany
lukas.bernhard@cispa.de

Nico Schiller
CISPA Helmholtz Center for
Information Security
Germany
nico.schiller@cispa.de

Moritz Schloegel
CISPA Helmholtz Center for
Information Security
Germany
moritz.schloegel@cispa.de

Nils Bars
CISPA Helmholtz Center for
Information Security
Germany
nils.bars@cispa.de

Thorsten Holz
CISPA Helmholtz Center for
Information Security
Germany
holz@cispa.de

ABSTRACT

A recent trend towards running more demanding web applications, such as video games or client-side LLMs, in the browser has led to the adoption of the WebGPU standard that provides a cross-platform API exposing the GPU to websites. This opens up a new attack surface: Untrusted web content is passed through to the GPU stack, which traditionally has been optimized for performance instead of security. Worsening the problem, most of WebGPU cannot be run in the tightly sandboxed process that manages other web content, which eases the attacker's path to compromising the client machine. Contrasting its importance, WebGPU shader processing has received surprisingly little attention from the automated testing community. Part of the reason is that shader translators expect highly structured and statically typed input, which renders typical fuzzing mutations ineffective. Complicating testing further, shader translation consists of a complex multi-step compilation pipeline, each stage presenting unique requirements and challenges.

In this paper, we propose DarthShader, the first language fuzzer that combines mutators based on an intermediate representation with those using a more traditional abstract syntax tree. The key idea is that the individual stages of the shader compilation pipeline are susceptible to different classes of faults, requiring entirely different mutation strategies for thorough testing. By fuzzing the full pipeline, we ensure that we maintain a realistic attacker model. In an empirical evaluation, we show that our method outperforms the state-of-the-art fuzzers regarding code coverage. Furthermore, an extensive ablation study validates our key design. DarthShader found a total of 39 software faults in all modern browsers—Chrome, Firefox, and Safari—that prior work missed. For 15 of them, the Chrome team assigned a CVE , acknowledging the impact of our results.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ACM Conference on Computer and Communications Security (CCS), http://dx.doi.org/10.1145/3658644.3690209.

CCS CONCEPTS

• **Security and privacy** → **Browser security**; *Systems security*; • **Computing methodologies** → *Graphics systems and interfaces.*

KEYWORDS

Fuzzing, Software Security, Browser Security, Graphics Shaders, WebGPU, WGSL

ACM Reference Format:
Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. 2024. DarthShader: Fuzzing WebGPU Shader Translators & Compilers. In *Proceedings of ACM Conference on Computer and Communications Security (CCS '24).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690209

1 INTRODUCTION

The internet and the web have been game changers in the past decades, enabling instant access to global news, constant connection with friends and acquaintances, and many types of new business models. Web browsers, in particular, play a crucial role in this ecosystem, as they are the most important applications to access the web for many users. However, the ubiquitous connectivity of the internet also enables adversaries with malicious intent, exposing users to potential threats as they navigate the web. A common security risk is memory safety violations [50], which have been the starting point for many successful attacks in the past.

As a result, we require fundamental, proactive measures to improve defenses against such threats and strengthen web browsers against various attack vectors. By using hardware-supported security features such as memory randomization (ASLR) and non-executable memory regions, web browsers can reduce the risk of exploits that attempt to execute arbitrary code. Moreover, rigorous testing needs to be performed on all browser components. This includes web APIs [16, 26] and JavaScript engines [21, 23, 39, 43, 54], given that they are often targeted due to their complexity and the fine-grained control they expose to adversaries. In addition, *sandboxing* is a crucial defense mechanism designed to prevent code from performing malicious actions or accessing sensitive data outside its intended scope [14, 37]. This technique enforces a strict separation between the content of different websites in different processes (called *site isolation* [41]) and most importantly between web content and the privileged components of the browser, e.g., those with access to the file system. Technically speaking, sandboxing is implemented by executing code of different sites in separate processes with restricted authorizations. Each process is confined by a security policy enforced at the operating system level, which

Takeaways

- GPU stack is exposed by WebGPU
- GPU stacks are a weak spot and lack in-depth security testing

DarthShader: Fuzzing WebGPU Shader Translators & Compilers

Lukas Bernhard
CISPA Helmholtz Center for
Information Security
Germany
lukas.bernhard@cispa.de

Nico Schiller
CISPA Helmholtz Center for
Information Security
Germany
nico.schiller@cispa.de

Moritz Schloegel
CISPA Helmholtz Center for
Information Security
Germany
moritz.schloegel@cispa.de

Nils Bars
CISPA Helmholtz Center for
Information Security
Germany
nils.bars@cispa.de

Thorsten Holz
CISPA Helmholtz Center for
Information Security
Germany
holz@cispa.de

ABSTRACT

A recent trend towards running more demanding web applications, such as video games or client-side LLMs, in the browser has led to the adoption of the WebGPU standard that provides a cross-platform API exposing the GPU to websites. This opens up a new attack surface: Untrusted web content is passed through to the GPU stack, which traditionally has been optimized for performance instead of security. Worsening the problem, most of WebGPU cannot be run in the tightly sandboxed process that manages other web content, which eases the attacker's path to compromising the client machine. Contrasting its importance, WebGPU shader processing has received surprisingly little attention from the automated testing community. Part of the reason is that shader translators expect highly structured and statically typed input, which renders typical fuzzing mutations ineffective. Complicating testing further, shader translation consists of a complex multi-step compilation pipeline, each stage presenting unique requirements and challenges.

In this paper, we propose DarthShader, the first language fuzzer that combines mutators based on an intermediate representation with those using a more traditional abstract syntax tree. The key idea is that the individual stages of the shader compilation pipeline are susceptible to different classes of faults, requiring entirely different mutation strategies for thorough testing. By fuzzing the full pipeline, we ensure that we maintain a realistic attacker model. In an empirical evaluation, we show that our method outperforms the state-of-the-art fuzzers regarding code coverage. Furthermore, an extensive ablation study validates our key design. DarthShader found a total of 39 software faults in all modern browsers—Chrome, Firefox, and Safari—that prior work missed. For 15 of them, the Chrome team assigned a CVE , acknowledging the impact of our results.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ACM Conference on Computer and Communications Security (CCS), http://dx.doi.org/10.1145/3658644.3690209.

CCS CONCEPTS

• **Security and privacy** → **Browser security**; *Systems security*; • **Computing methodologies** → *Graphics systems and interfaces.*

KEYWORDS

Fuzzing, Software Security, Browser Security, Graphics Shaders, WebGPU, WGSL

ACM Reference Format:
Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. 2024. DarthShader: Fuzzing WebGPU Shader Translators & Compilers. In *Proceedings of ACM Conference on Computer and Communications Security (CCS '24)*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690209

1 INTRODUCTION

The internet and the web have been game changers in the past decades, enabling instant access to global news, constant connection with friends and acquaintances, and many types of new business models. Web browsers, in particular, play a crucial role in this ecosystem, as they are the most important applications to access the web for many users. However, the ubiquitous connectivity of the internet also enables adversaries with malicious intent, exposing users to potential threats as they navigate the web. A common security risk is memory safety violations [50], which have been the starting point for many successful attacks in the past.

As a result, we require fundamental, proactive measures to improve defenses against such threats and strengthen web browsers against various attack vectors. By using hardware-supported security features such as memory randomization (ASLR) and non-executable memory regions, web browsers can reduce the risk of exploits that attempt to execute arbitrary code. Moreover, rigorous testing needs to be performed on all browser components. This includes web APIs [16, 26] and JavaScript engines [21, 23, 39, 43, 54], given that they are often targeted due to their complexity and the fine-grained control they expose to adversaries. In addition, *sandboxing* is a crucial defense mechanism designed to prevent code from performing malicious actions or accessing sensitive data outside its intended scope [14, 37]. This technique enforces a strict separation between the content of different websites in different processes (called *site isolation* [41]) and most importantly between web content and the privileged components of the browser, e.g., those with access to the file system. Technically speaking, sandboxing is implemented by executing code of different sites in separate processes with restricted authorizations. Each process is confined by a security policy enforced at the operating system level, which

Takeaways

- GPU stack is exposed by WebGPU
- GPU stacks are a weak spot
  and lack in-depth security testing
- Combination of IR and AST fuzzing
  exposes multitude of security issues

DarthShader: Fuzzing WebGPU Shader Translators & Compilers

Lukas Bernhard
CISPA Helmholtz Center for
Information Security
Germany
lukas.bernhard@cispa.de

Nico Schiller
CISPA Helmholtz Center for
Information Security
Germany
nico.schiller@cispa.de

Moritz Schloegel
CISPA Helmholtz Center for
Information Security
Germany
moritz.schloegel@cispa.de

Nils Bars
CISPA Helmholtz Center for
Information Security
Germany
nils.bars@cispa.de

Thorsten Holz
CISPA Helmholtz Center for
Information Security
Germany
holz@cispa.de

ABSTRACT

A recent trend towards running more demanding web applications, such as video games or client-side LLMs, in the browser has led to the adoption of the WebGPU standard that provides a cross-platform API exposing the GPU to websites. This opens up a new attack surface: Untrusted web content is passed through to the GPU stack, which traditionally has been optimized for performance instead of security. Worsening the problem, most of WebGPU cannot be run in the tightly sandboxed process that manages other web content, which eases the attacker's path to compromising the client machine. Contrasting its importance, WebGPU shader processing has received surprisingly little attention from the automated testing community. Part of the reason is that shader translators expect highly structured and statically typed input, which renders typical fuzzing mutations ineffective. Complicating testing further, shader translation consists of a complex multi-step compilation pipeline, each stage presenting unique requirements and challenges.

In this paper, we propose DarthShader, the first language fuzzer that combines mutators based on an intermediate representation with those using a more traditional abstract syntax tree. The key idea is that the individual stages of the shader compilation pipeline are susceptible to different classes of faults, requiring entirely different mutation strategies for thorough testing. By fuzzing the full pipeline, we ensure that we maintain a realistic attacker model. In an empirical evaluation, we show that our method outperforms the state-of-the-art fuzzers regarding code coverage. Furthermore, an extensive ablation study validates our key design. DarthShader found a total of 39 software faults in all modern browsers—Chrome, Firefox, and Safari—that prior work missed. For 15 of them, the Chrome team assigned a CVE , acknowledging the impact of our results.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ACM Conference on Computer and Communications Security (CCS), http://dx.doi.org/10.1145/3658644.3690209.

CCS CONCEPTS

• **Security and privacy** → **Browser security**; *Systems security*; • **Computing methodologies** → *Graphics systems and interfaces.*

KEYWORDS

Fuzzing, Software Security, Browser Security, Graphics Shaders, WebGPU, WGSL

ACM Reference Format:
Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. 2024. DarthShader: Fuzzing WebGPU Shader Translators & Compilers. In *Proceedings of ACM Conference on Computer and Communications Security (CCS '24)*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690209

1 INTRODUCTION

The internet and the web have been game changers in the past decades, enabling instant access to global news, constant connection with friends and acquaintances, and many types of new business models. Web browsers, in particular, play a crucial role in this ecosystem, as they are the most important applications to access the web for many users. However, the ubiquitous connectivity of the internet also enables adversaries with malicious intent, exposing users to potential threats as they navigate the web. A common security risk is memory safety violations [50], which have been the starting point for many successful attacks in the past.

As a result, we require fundamental, proactive measures to improve defenses against such threats and strengthen web browsers against various attack vectors. By using hardware-supported security features such as memory randomization (ASLR) and non-executable memory regions, web browsers can reduce the risk of exploits that attempt to execute arbitrary code. Moreover, rigorous testing needs to be performed on all browser components. This includes web APIs [16, 26] and JavaScript engines [21, 23, 39, 43, 54], given that they are often targeted due to their complexity and the fine-grained control they expose to adversaries. In addition, *sandboxing* is a crucial defense mechanism designed to prevent code from performing malicious actions or accessing sensitive data outside its intended scope [14, 37]. This technique enforces a strict separation between the content of different websites in different processes (called *site isolation* [41]) and most importantly between web content and the privileged components of the browser, e.g., those with access to the file system. Technically speaking, sandboxing is implemented by executing code of different sites in separate processes with restricted authorizations. Each process is confined by a security policy enforced at the operating system level, which

Takeaways

- GPU stack is exposed by WebGPU
- GPU stacks are a weak spot and lack in-depth security testing
- Combination of IR and AST fuzzing exposes multitude of security issues

github.com/wgslfuzz/darthshader

DarthShader: Fuzzing WebGPU Shader Translators & Compilers

Lukas Bernhard
CISPA Helmholtz Center for
Information Security
Germany
lukas.bernhard@cispa.de

Nico Schiller
CISPA Helmholtz Center for
Information Security
Germany
nico.schiller@cispa.de

Moritz Schloegel
CISPA Helmholtz Center for
Information Security
Germany
moritz.schloegel@cispa.de

Nils Bars
CISPA Helmholtz Center for
Information Security
Germany
nils.bars@cispa.de

Thorsten Holz
CISPA Helmholtz Center for
Information Security
Germany
holz@cispa.de

ABSTRACT

A recent trend towards running more demanding web applications, such as video games or client-side LLMs, in the browser has led to the adoption of the WebGPU standard that provides a cross-platform API exposing the GPU to websites. This opens up a new attack surface: Untrusted web content is passed through to the GPU stack, which traditionally has been optimized for performance instead of security. Worsening the problem, most of WebGPU cannot be run in the tightly sandboxed process that manages other web content, which eases the attacker's path to compromising the client machine. Contrasting its importance, WebGPU shader processing has received surprisingly little attention from the automated testing community. Part of the reason is that shader translators expect highly structured and statically typed input, which renders typical fuzzing mutations ineffective. Complicating testing further, shader translation consists of a complex multi-step compilation pipeline, each stage presenting unique requirements and challenges.

In this paper, we propose DarthShader, the first language fuzzer that combines mutators based on an intermediate representation with those using a more traditional abstract syntax tree. The key idea is that the individual stages of the shader compilation pipeline are susceptible to different classes of faults, requiring entirely different mutation strategies for thorough testing. By fuzzing the full pipeline, we ensure that we maintain a realistic attacker model. In an empirical evaluation, we show that our method outperforms the state-of-the-art fuzzers regarding code coverage. Furthermore, an extensive ablation study validates our key design. DarthShader found a total of 39 software faults in all modern browsers—Chrome, Firefox, and Safari—that prior work missed. For 15 of them, the Chrome team assigned a CVE , acknowledging the impact of our results.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ACM Conference on Computer and Communications Security (CCS), http://dx.doi.org/10.1145/3658644.3690209.

CCS CONCEPTS

• **Security and privacy** → **Browser security**; *Systems security*; • **Computing methodologies** → *Graphics systems and interfaces.*

KEYWORDS

Fuzzing, Software Security, Browser Security, Graphics Shaders, WebGPU, WGSL

ACM Reference Format:
Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. 2024. DarthShader: Fuzzing WebGPU Shader Translators & Compilers. In *Proceedings of ACM Conference on Computer and Communications Security (CCS '24)*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690209

1 INTRODUCTION

The internet and the web have been game changers in the past decades, enabling instant access to global news, constant connection with friends and acquaintances, and many types of new business models. Web browsers, in particular, play a crucial role in this ecosystem, as they are the most important applications to access the web for many users. However, the ubiquitous connectivity of the internet also enables adversaries with malicious intent, exposing users to potential threats as they navigate the web. A common security risk is memory safety violations [50], which have been the starting point for many successful attacks in the past.

As a result, we require fundamental, proactive measures to improve defenses against such threats and strengthen web browsers against various attack vectors. By using hardware-supported security features such as memory randomization (ASLR) and non-executable memory regions, web browsers can reduce the risk of exploits that attempt to execute arbitrary code. Moreover, rigorous testing needs to be performed on all browser components. This includes web APIs [16, 26] and JavaScript engines [21, 23, 39, 43, 54], given that they are often targeted due to their complexity and the fine-grained control they expose to adversaries. In addition, *sandboxing* is a crucial defense mechanism designed to prevent code from performing malicious actions or accessing sensitive data outside its intended scope [14, 37]. This technique enforces a strict separation between the content of different websites in different processes (called *site isolation* [41]) and most importantly between web content and the privileged components of the browser, e.g., those with access to the file system. Technically speaking, sandboxing is implemented by executing code of different sites in separate processes with restricted authorizations. Each process is confined by a security policy enforced at the operating system level, which