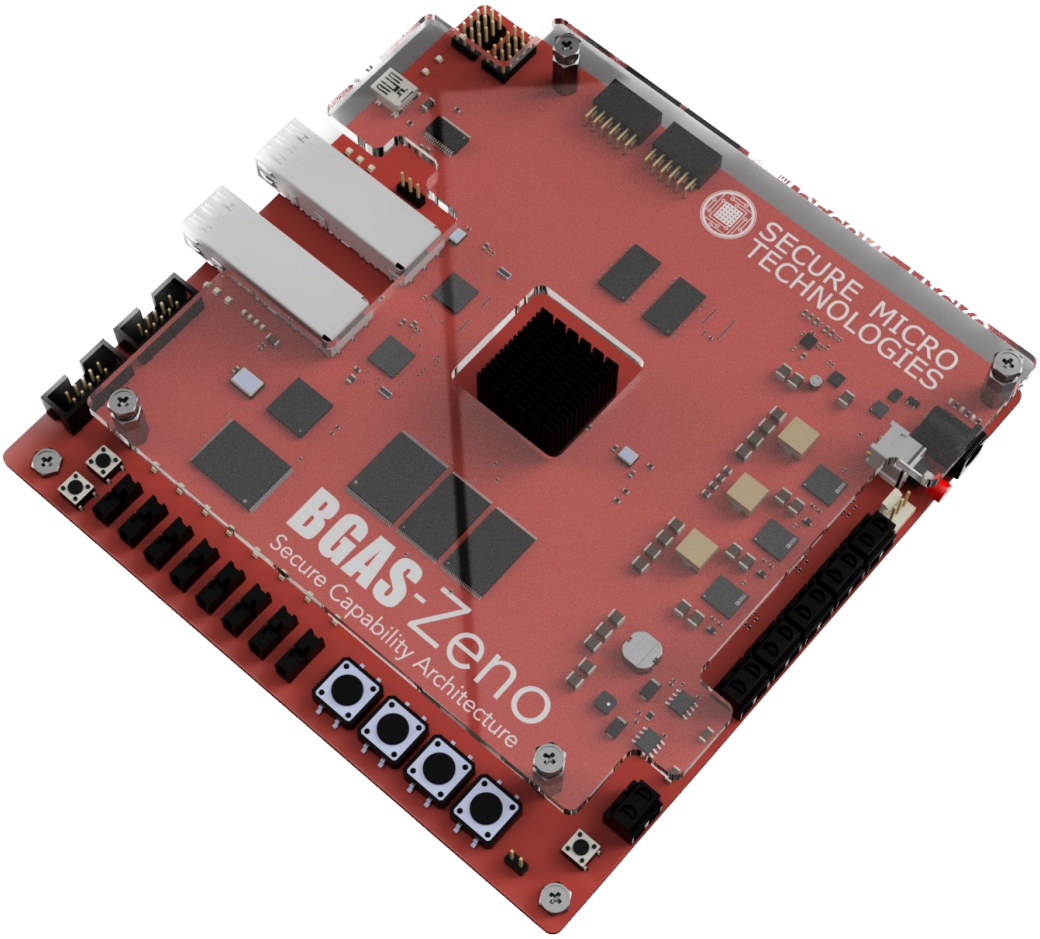
SECURE MICRO TECHNOLOGIES

BGAS-Zeno: Secure Capability Platform Manual

Version 1.1



info@securemicro.com

April 5, 2024

Contents

1 Introduction

1. Scope and Intended Audience

This manual describes the BGAS-Zeno capability software development board and how to use it. The intended audience for this manual is software developers interested in capability-aware application development. For more detailed information about the BGAS-Zeno ISA and architecture, see the BGAS-Zeno ISA specification. Visit the BGAS-Zeno Homepage for more information about the BGAS-Zeno Capability Platform.

2. Platform Description

The BGAS-Zeno Platform provides a complete hardware/software system stack for capability-aware application development. The BGAS-Zeno IDE provides a development environment to write applications for the BGAS-Zeno system. The IDE includes the capability-aware Zeno-llvm tool-chain to compile executables targeting the BGAS-Zeno architecture. Executables created in the IDE are downloaded to the board and run as user-mode applications in the Linux operating system (OS). The BGAS-Zeno CPU executes capability-aware applications, enforcing capability access permissions in hardware. Figure 1 describes the layout of the BGAS-Zeno board.

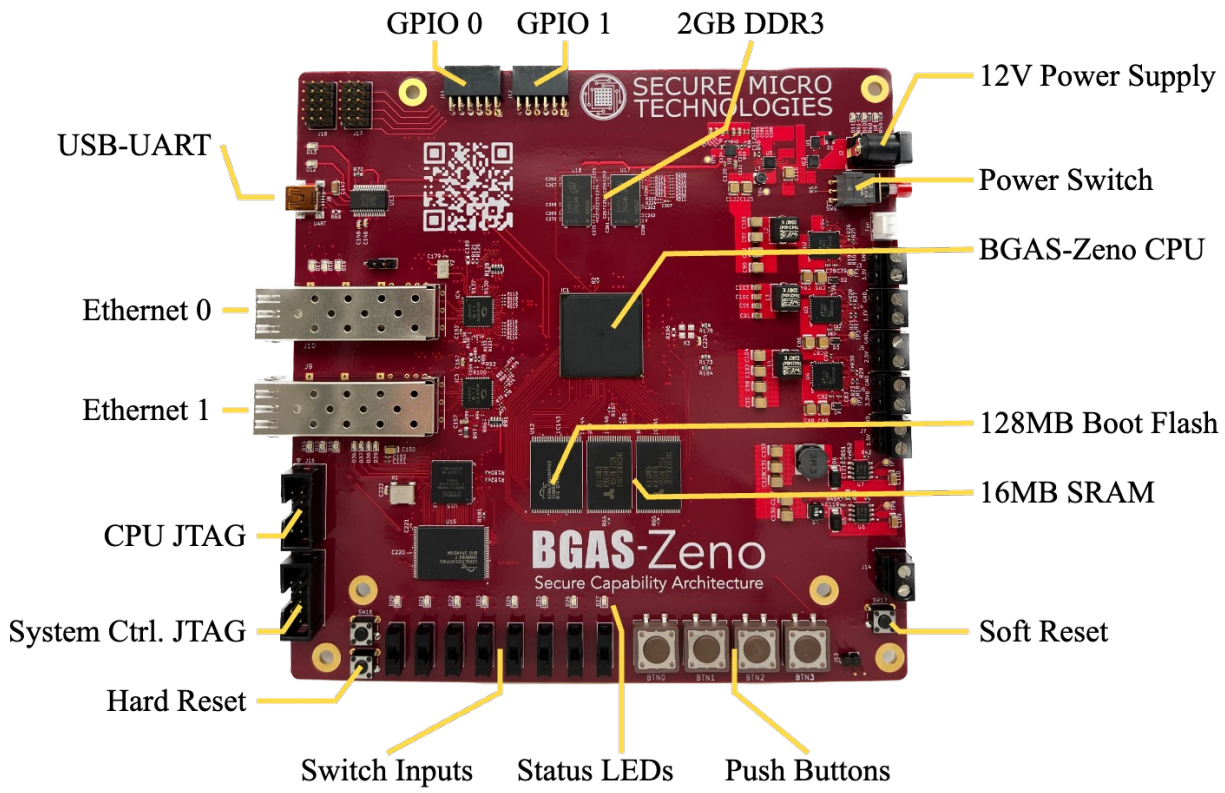


Figure 1: BGAS-Zeno Secure Software Development Board layout.

2 Getting Started with the BGAS-Zeno Board

1. System Boot-up

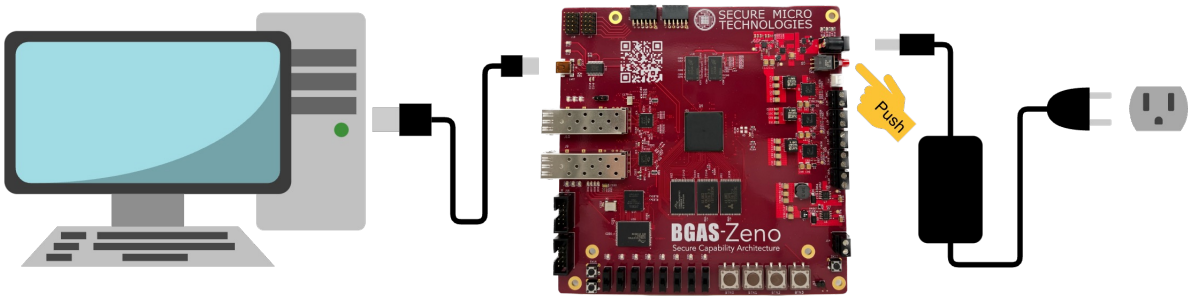


Figure 2: Basic connections to get started with the BGAS-Zeno board.

1. Connect the 12V DC power adapter to the barrel jack connector of the BGAS-Zeno board. Push the red power switch next to the barrel jack to power on the board. LEDs next to the power jack indicate power for each of the 3.3V and 2.5V rails.

2. Connect the included USB Mini-B cable to the USB Mini-B connector on the board. Plug the other end into your workstation. On Ubuntu 22.04, the board will appear as a serial device such as `/dev/ttyUSB0`.

3. On your workstation, Open the Secure Micro Technologies BGAS-Zeno IDE to connect to the BGAS-Zeno board. Alternatively, you may use an application such as `minicom` to communicate with the board over the serial port.

4. Wait up to 180 seconds for the BGAS-Zeno board to complete the boot process. Once complete, the board will print a welcome banner and provide access to a Linux command prompt with support for common utilities and built-in example applications. The output of a complete boot sequence is shown in Figure 3.

```
[    8.194700] mousedev: PS/2 mouse device common for all mice
[    8.205592] cpuidle-riscv-sbi: HSM suspend not available
[    8.211947] sdhci: Secure Digital Host Controller Interface driver
[    8.212468] sdhci: Copyright(c) Pierre Ossman
[    8.214607] sdhci-pltfm: SDHCI platform and OF driver helper
[    8.276697] debug_vm_pgtable: [debug_vm_pgtable         ]: Validating architecture page table helpers
[    8.415736] Legacy PMU implementation is available
[   41.585796] Freeing unused kernel image (initmem) memory: 11388K
[   41.590514] Run /init as init process

Boot took 42.05 seconds

#------------------------------------------------------------#
#    Welcome to the BGAS-Zeno Secure Capability Platform!    #
#                Secure Micro Technologies                   #
#                  info@securemicro.com                      #
#------------------------------------------------------------#

/ #
```

Figure 3: The BGAS-Zeno system output after completing the boot process.

2.2 Executing Built-in Examples

The BGAS-Zeno system includes several built-in example applications to demonstrate capability features. The following directory tree shows a subset of the Linux directory structure on the BGAS-Zeno system with the built-in example applications. Use the Linux `cd` and `ls` commands to navigate the directory structure and execute the examples.

```
/root
├── capability_example.exe
├── cpi.exe
├── hello_world.exe
└── benchmarks
    ├── riscv
    │   ├── double_free.c
    │   ├── double_free.exe
    │   ├── oob_read.c
    │   ├── oob_read.exe
    │   ├── oob_write.c
    │   └── oob_write.exe
    └── zeno
        ├── double_free.c
        ├── double_free.exe
        ├── oob_read.c
        ├── oob_read.exe
        ├── oob_write.c
        └── oob_write.exe
```

- The `/root` directory serves as the home directory for the built-in examples.
- The `capability_example.exe` demonstrates capability creation, derivation, and revocation.
- The `cpi.exe` computes the Cycles-per-Instruction and prints the number of cycles and instructions committed since boot.
- The `hello_world.exe` prints a “hello world” message.
- The `benchmarks` directory includes memory vulnerability benchmarks with and without capability protections.
- Inside `benchmarks`, the `riscv` directory includes the non-capability-aware versions of the benchmarks.
- The `zeno` directory includes versions of the benchmarks compiled with capability support.
- The `double_free.exe` executables demonstrate a temporal memory safety vulnerability with and without capability support.
- The `oob_read.exe` executables demonstrate a buffer out-of-bounds read vulnerability with and without capability support.
- The `oob_write.exe` executables demonstrate a buffer out-of-bounds write vulnerability with and without capability support.

After the BGAS-Zeno system has booted, executing the `capability-example.exe` will produce the following output:

```
/root # ./capability_example.exe
#-------------------------------------------------------#
# Welcome to the BGAS-Zeno Secure Capability Platform! #
#              Secure Micro Technologies                #
#                info@securemicro.com                   #
#-------------------------------------------------------#
Executing: Capability Example App...


[C_CREATE] C: 0x0000000000000001
    Min:  0x0000000000000000
    Max:  0x0000000000000400
    Perm: 0x0000000000000007

[C_CREATE] C: 0x0000000000000002
    Min:  0x0000000000000400
    Max:  0x0000000000000800
    Perm: 0x0000000000000004

[C_DERIVE] C: 0x0000000000000003
    P:    0x0000000000000001
    Min:  0x0000000000000000
    Max:  0x0000000000000100
    Perm: 0x0000000000000006

[C_DERIVE] C: 0x0000000000000004
    P:    0x0000000000000003
    Min:  0x0000000000000000
    Max:  0x0000000000000100
    Perm: 0x0000000000000004

[C_REVOKE] C: 0x0000000000000001

[C_REVOKE] C: 0x0000000000000003

[C_REVOKE] C: 0x0000000000000004

[C_REVOKE] C: 0x0000000000000002
/root #
```

2.3 BGAS-Zeno IDE

The BGAS-Zeno IDE provides a complete environment to develop capability-aware applications and execute them on the BGAS-Zeno system. A screenshot of the IDE is shown in Figure 4. The IDE includes three main windows: (1) the code-editor window for writing applications, (2) the share window for downloading executables to the BGAS-Zeno system. and (3) the terminal window for interacting with the system and executing applications.

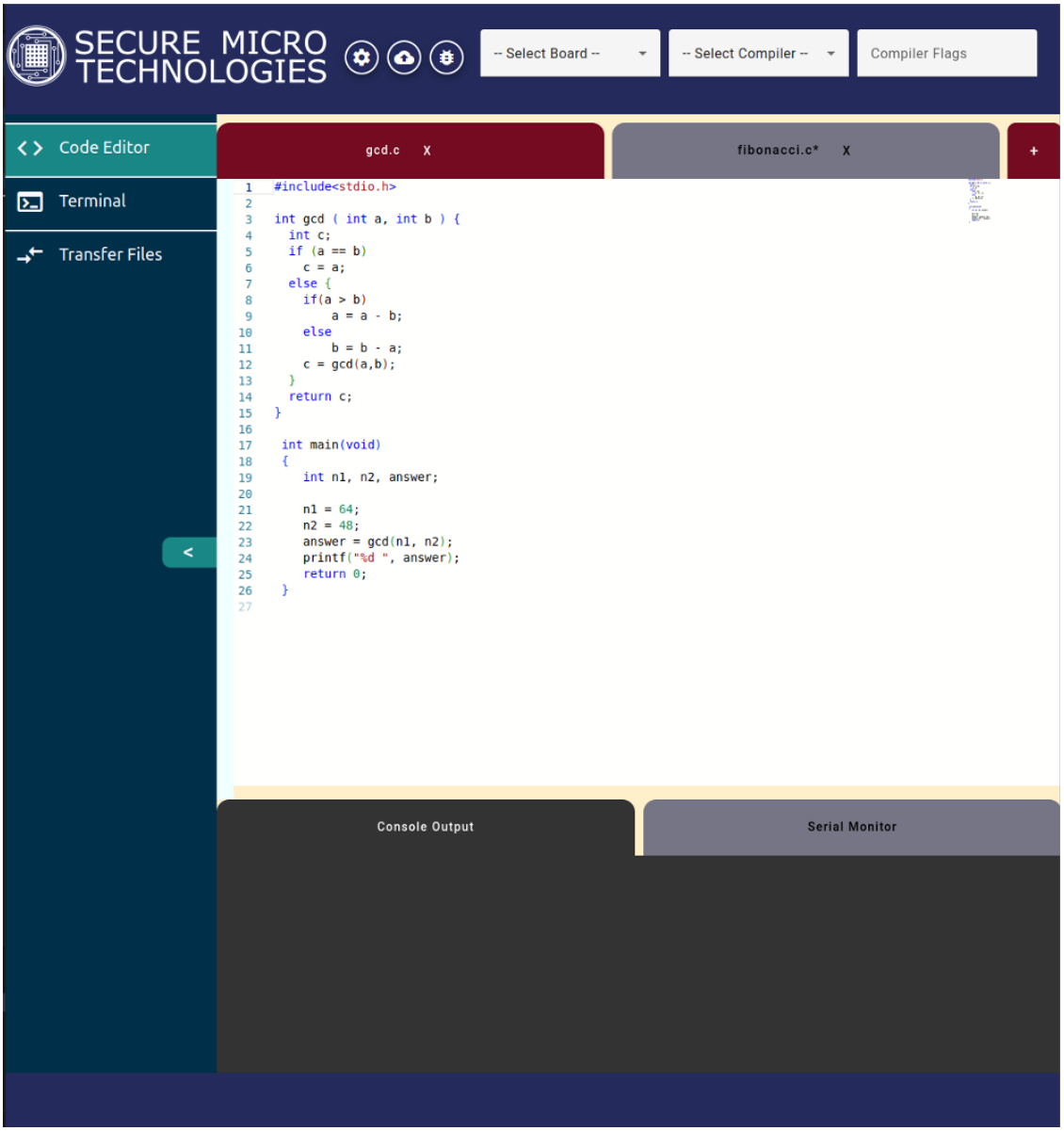


Figure 4: BGAS-Zeno IDE code-editor window.

2.3.1 Code-Editor Window

The menu bar of the code editor window, shown in Figure 5, provides several tools and configuration options. Each menu item is described in detail below.

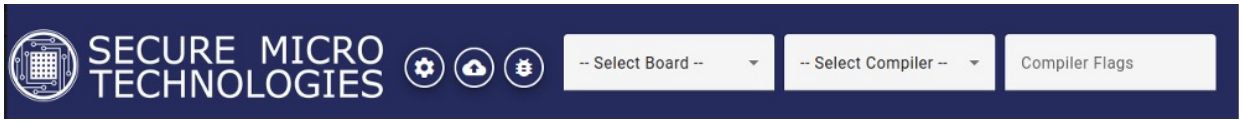
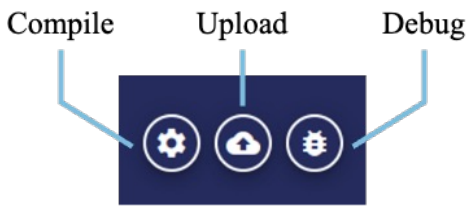
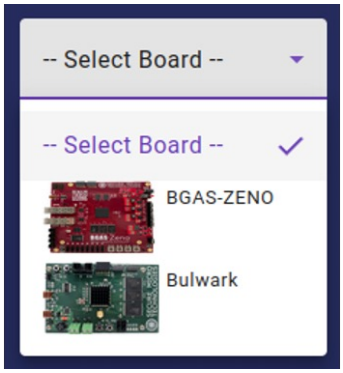


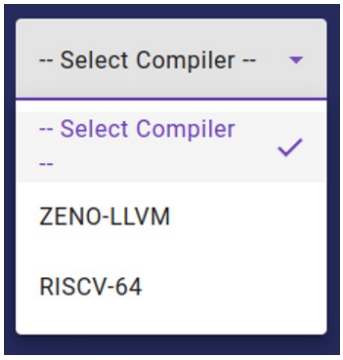
Figure 5: BGAS-Zeno IDE code-editor window header bar.



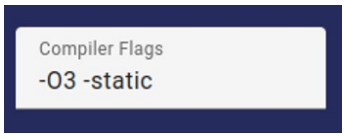
Click the `Compile` button to compile the application in the code-editor window using the selected compiler. Compiler output is shown in the console output tab. To transfer executables to the selected BGAS-Zeno system, click the upload button. The compiled executable will automatically be written to /root.



Select BGAS-Zeno in the board selection drop-down menu to compile binaries for the BGAS-Zeno system.

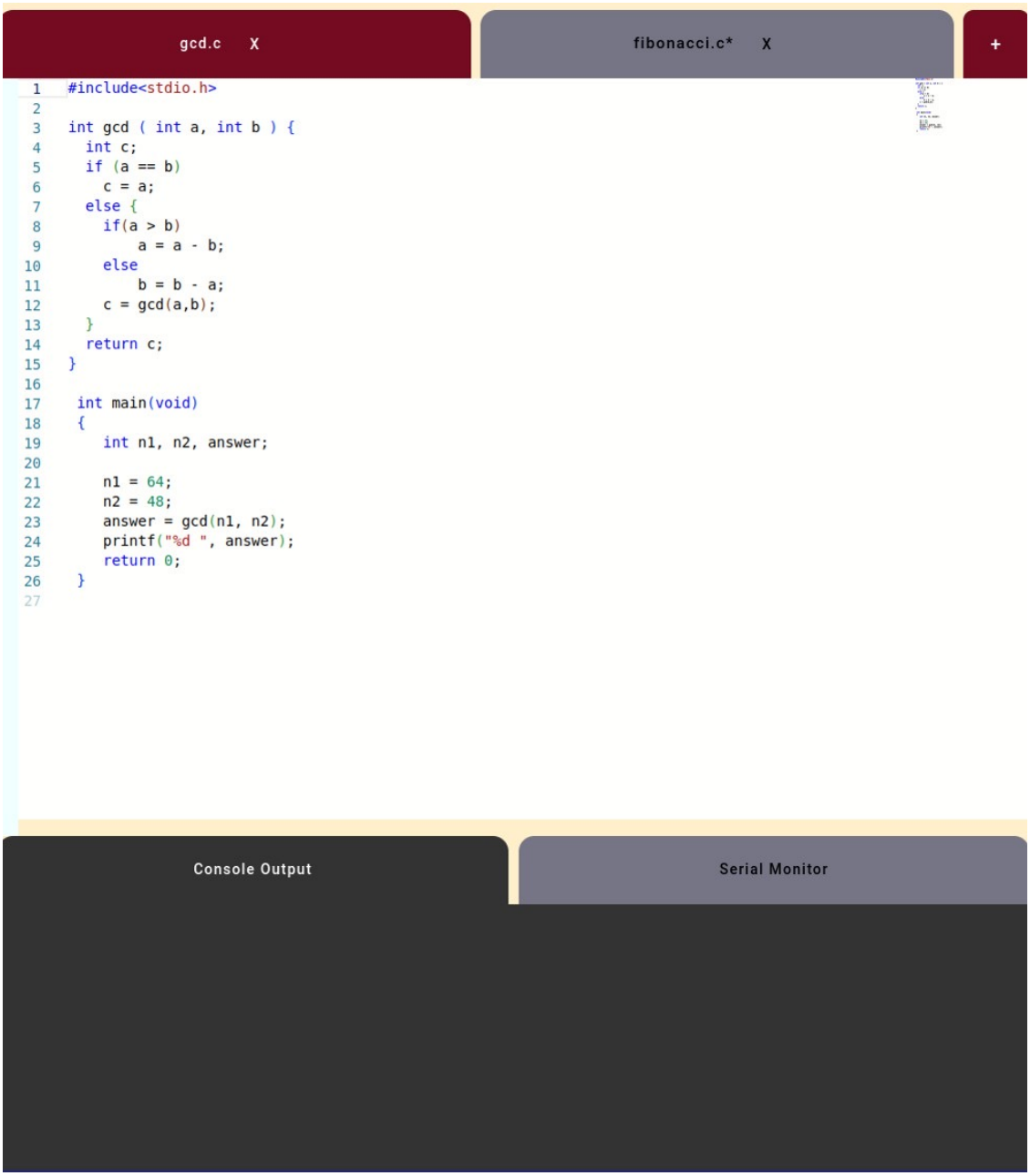


Use the compiler selection drop-down menu to select between the Zeno-LLVM compiler with capability support and the RISC-V GNU GCC compiler without capability support.



Optionally enter additional flags to pass to the compiler during compilation.

The editor window provides an editable text area to write code. The editor supports copy/paste, foldable function segments and a minimap providing an overview of the entire code. Dynamic tabs in the code editor allow users to edit up to five programs at once. Double click on the tab to rename it. Clicking the X on the tab closes and deletes the tab (minimum tabs allowed is 1). The + button at the right end adds new tabs (maximum tabs allowed are 5). Content written in the editor window is autosaved every five seconds. The * mark in a tab name indicates that the content of the tab is unsaved. The console output tab shows compiler output, including errors and warnings. The serial output tab shows the program output when executed on the selected board.



2.3.2 Terminal Window

The terminal window provides an interface to interact with the BGAS-Zeno system. Figure 6 shows the menu bar visible in the terminal window. Each menu item is described in detail below.

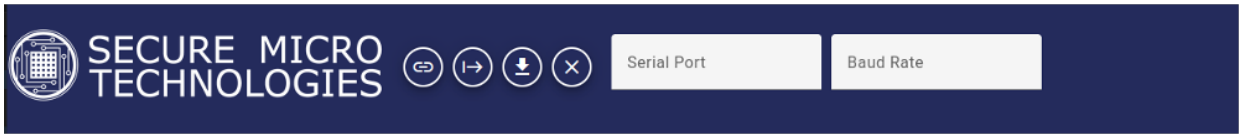
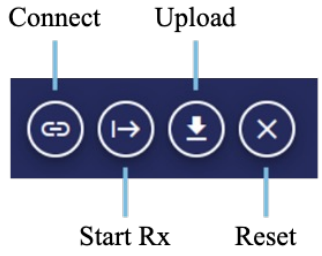
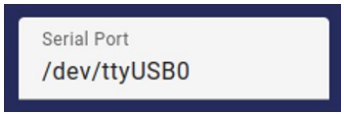


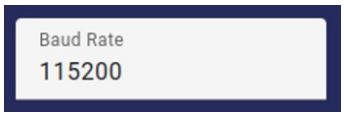
Figure 6: BGAS-Zeno IDE terminal window header bar.



Use the connect button to connect to the BGAS-Zeno system's serial port. Click the Start Receiver Button (Start Rx) to initialize the BGAS-Zeno system for file transfers. The Receive Button is used to start receiving data from the BGAS-Zeno system. Clicking the Reset Button disconnects the BGAS-Zeno system and closes the receiver.



Enter the serial port the BGAS-Zeno board is connected to in the serial port input field.



Enter "115200" as the Baud Rate when connecting to the BGAS-Zeno system.

Figure 7 shows the BGAS-Zeno IDE terminal window. Serial data received from the BGAS-Zeno system is displayed as it is received by the IDE. Commands can be written after the ?¿ symbol. To send the command to the board, press the enter key on the keyboard. Then, Data entered in the command box is transmitted to the BGAS-Zeno system. Resulting output will be shown on the terminal screen above the ?¿ symbol.

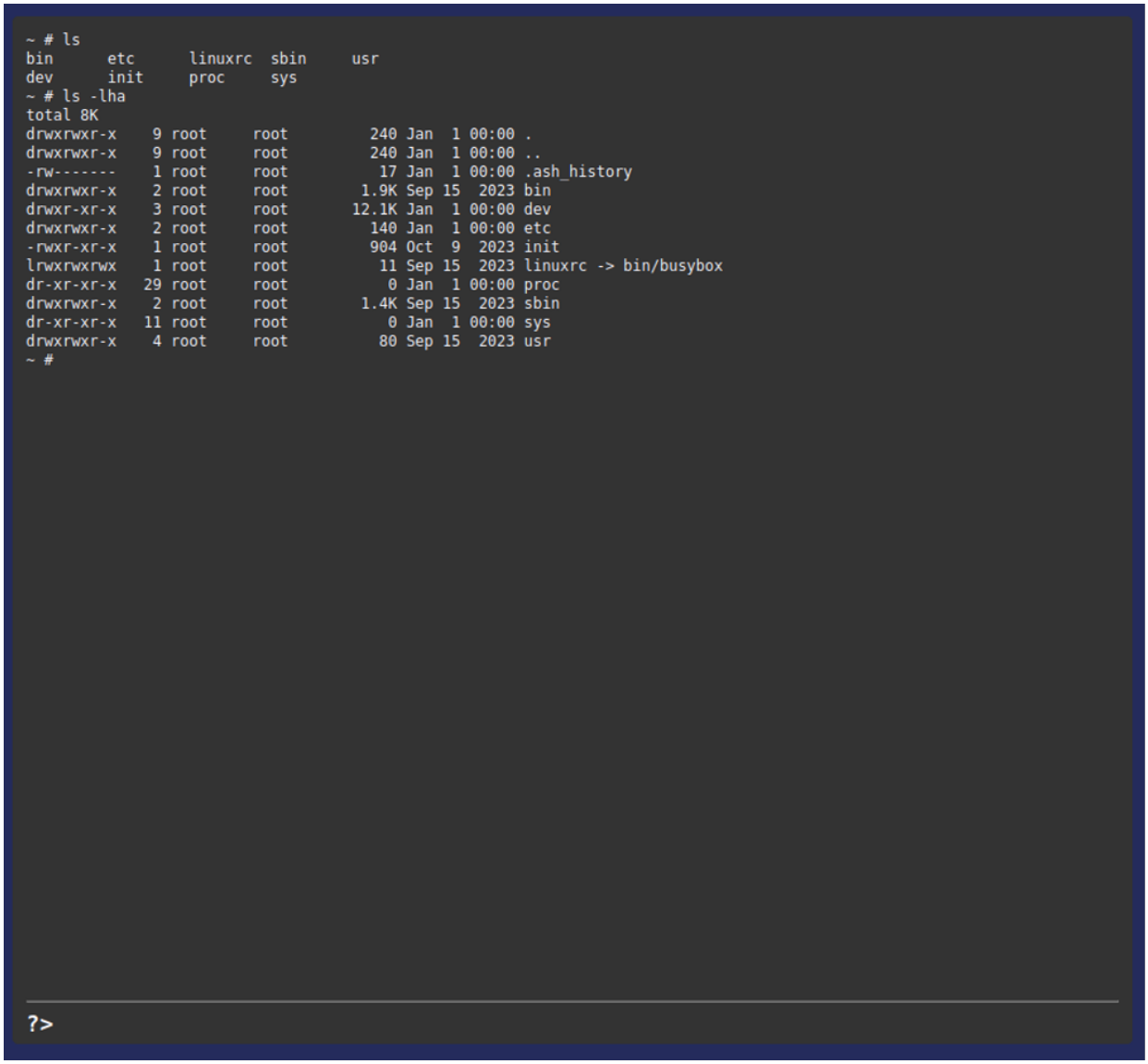


Figure 7: BGAS-Zeno IDE terminal window.

3 Capability-Aware Software Development

1. Capability Library and API

A user-level capability library (`libcap_runtime.a`) provides a C language Application Programming Interface (API) to develop capability-aware software applications on the BGAS-Zeno architecture. API functions provide access to hardware features not directly supported by non-capability-aware system software stacks, such as RISC-V OpenSBI, Linux and the GNU GCC compiler. The capability library provides all the support necessary to develop user-level capability-aware applications for the BGAS-Zeno architecture. Table 1 describes each API function and its C language arguments and return type.

Table 1: BGAS-Zeno User-Level API.

API Function	Description
`Capability myCapability;`	Capability declaration. With The Zeno compiler, The declared variable will behave the same as a void pointer.
`Capability_id myCapability_id;`	Capability ID declaration. Use the declared variable to store a capability's capability ID, i.e the value stored in the capability register file.

`Capability c_create(`
`  unsigned long min,`
`  unsigned long max,`
`  unsigned long perm`
`);`
	Create Capability function prototype. Capability access permissions and min/max address bounds are passed as arguments. The min and max arguments are 64-bit unsigned integers representing the pointer portion of a memory reference. The perm argument is 64-bits of permission values. Currently only read (bit 2), write (bit 1), and execute (bit 0) permissions are supported. If successful, metadata structures are updated and a new capability is returned.

`Capability c_derive(`
`  unsigned long min,`
`  unsigned long max,`
`  unsigned long perm,`
`  Capability c_parent`
`);`
	Derive Capability function prototype. The parent capability, capability access permissions and min/max address bounds are passed as arguments using the same format as the `c_create` function. Requested access permissions are validated to ensure monotonically decreasing permissions. If the validation is successful, a new capability is created and added to the parent capability's child list. Metadata structures are updated with the new capability metadata and the new capability is returned.

Table 1: BGAS-Zeno User-Level API. (Continued)

API Function	Description
`void c_revoke(Capability c);`	Revoke Capability function prototype. Recursively revoke the given capability and all of its children. The access permissions are removed from metadata structures, preventing future access with the capability. However, the capability IDs in memory and on-chip are not invalidated. Software may still attempt memory accesses with them, however all accesses will fail because permissions associated with that capability have been removed.
`void c_revoke(Capability c);`	Get Capability ID function prototype. A helper function to get the capability ID of a given capability.

`Capability convert_id(`
`  Capability_id id`
`);`
	Convert Capability ID function prototype. A helper function to move the given capability ID into the capability register file. A capability with the given ID is returned. Note that if the given ID is invalid, the returned capability will also be invalid.

`Capability merge_ptr_id(`
`  void *ptr,`
`  Capability_id id`
`);`
	Merge pointer and capability ID into a single capability. This function allows a programmer to associate an arbitrary pointer with a capability ID. While any pointer value can be given, only pointer values within the capability's access permission range will be able to access memory.

`void capability_init();`	Helper function to setup capability support in a user level application executing in a non-capability-aware operating system.
`void capability_exit();`	Helper function to clean up capability state in a user level application executing in a non-capability-aware operating system. This function also clears the capability register file to prevent non-capability-aware applications and operating systems from accidentally using stale capability values.
`void clear_capabilities();`	Helper function to clear the capability register file. A capability with full system access is written to each capability register so non-capability aware code may execute.

`void save_capabilities(`
`  Capability_id *c`
`);`
	Helper function to save the capability ID in each capability register. The argument `c` is a pointer to an array of 32 capability ID values.

Table 1: BGAS-Zeno User-Level API. (Continued)

API Function	Description
`void restore_capabilities( Capability_id *c );`	Helper function to restore a capability ID value to each capability register. The argument `c` is a pointer to an array of 32 capability ID values to be written to the capability register file.
`void print_capabilities();`	Helper function to restore print the value of each capability ID in the capability register file.
`int zeno_printf( const char *format, ... );`	Zeno Printf function prototype. This function provides a Capability-aware implementation of the `printf` function. The function takes the same arguments as `printf`. Currently only a subset of the format specifiers are supported, including `"%d"`, `"%u"`, `"%c"`, `"%s"`, and `"%l"`. Other format specifiers will print an error instead of a formatted value.

The Zeno-llvm compiler provides support for 128-bit pointers formed by BGAS-Zeno capabilities. A customized ABI is necessary to support function calls with BGAS-Zeno 128-bit capability memory references. Compiling appliations in the BGAS-Zeno IDE will automatically target the BGAS-Zeno ABI and link binaries with the capability library.

The capability library provides limted support for capabiltiy-aware implementation or wrappers of C standard library features (e.g. `zeno_printf()`). Calling non-capability-aware library functions from capability-aware functions will likely result in capability access violations, terminating the application. Wrapers for non-capability aware code can be created with the `save_capability()`, `clear_capability()`, and `restore_capability()` functions.

3.2 Application Example 1

The application in Listing 1 demonstrates manual capability creation and revocation. Manual control allows a programmer to reduce access permissions associated with a capability, for example, shrinking the access bounds or removing write permissions.

```
1 #include "cap_user.h"
2 #define ARR_SIZE 16
3
4 int arr[ARR_SIZE];
5 int *arr_ptr;
6 int sum = 0;
7
8 int main() {
```

```
9    capability_init();
10
11   arr_ptr = arr;
12   int *arr_full = (int *)c_create(arr_ptr, arr_ptr+ARR_SIZE, 6);
13   arr_full = (int *)merge_ptr_id(arr_ptr, get_id(arr_full));
14
15   for(int i=0; i<ARR_SIZE; i++) {
16     arr_full[i] = i;
17   }
18
19   for(int i=0; i<ARR_SIZE; i++) {
20     sum += arr_full[i];
21   }
22
23   /* Attempt to write out-of-bounds */
24   /* arr_full[ARR_SIZE] = -1;      */
25
26   c_revoke(arr_full);
27   capability_exit();
28   return sum;
29 }
```

Listing 1: Capability Creation Example.

The following code snippets step through the example application from Listing 1 to highlight important aspects of capability-aware application development on the BGAS-Zeno platform.

```
1 #include "cap_user.h"
```

Include the `cap_user.h` header file in each application to define the capability library function prototypes described in Table 1.

```
4 int arr[ARR_SIZE];
5 int *arr_ptr;
```

Line 4 of the program declares an integer array `arr`. By default, the Zeno-llvm compiler will use the capability initialized by the OS for global variables like `arr`. For operating systems that are not capability-aware, such as Linux, a capability with full access to the application's virtual address space is used. Line 5 creates a pointer `arr_ptr` to point to the array `arr`. The `arr_ptr` is passed as a function argument instead of `arr` to prevent compiler warnings.

```
8 int main() {
9   capability_init();
```

When developing capability-aware applications for non-capability-aware operating systems, call the `capability_init()` function before using any capability features.

```
12   int *arr_full = (int *)c_create(arr_ptr, arr_ptr+ARR_SIZE, 6);
```

To support capability-aware applications in operating systems that are not capability-aware, programmers may manually create capabilities. Line 12 creates a capability with read/write permissions and access bounds that match the size of the array. The `c_create()` function returns a capability (i.e. void pointer) with a valid capability ID and a `NULL` pointer value.

```
13   arr_full = (int *)merge_ptr_id(arr_ptr, get_id(arr_full));
```

To set a useful capability pointer value, call the `merge_ptr_id()` function to associate the capability ID and pointer value. The compiler will ensure the capability ID and pointer value in the returned capability are used together for future memory accesses.

```
26   c_revoke(arr_full);
```

When a capability is no longer needed, it may be revoked with a call to `c_revoke()`. Revoking a capability will also recursively revoke all capabilities derived from the original revoked capability.

```
27   capability_exit();
28   return sum;
29 }
```

Capability-aware applications must call the `capability_exit()` function before returning from `main()` to clean up capability state that may interfere with a non-capability-aware operating system. If executing in a capability-aware OS, the OS will clean up the capability state instead.

Executing the example application produces the following output. Print statements for each capability create and revoke are generated automatically by firmware to assist with debugging. Executing `echo $?` displays the return value of the program.

```
/root # ./example_app1.exe

[C_CREATE] C: 0x0000000000000001
    Min:  0x0000000000013080
    Max:  0x00000000000130C0
    Perm: 0x0000000000000006

[C_REVOKE] C: 0x0000000000000001
/root # echo $?
120
/root #
```

3.3 Application Example 2

The following code represents a complete out-of-bounds read application example, demonstrating capabilities catching a buffer overflow error. In this example, capabilities are automatically created with by the capability library function `zeno_malloc()`

```
1 #include "cap_user.h"
```

```
2
3  int main() {
4    capability_init();
5
6    zeno_printf("\nStarting test: OOB Read\n");
7
8    int test_status = 1;
9
10   char *public      = (char *)zeno_malloc(6);
11   zeno_strcpy(public, "public");
12
13   char *private     = (char *)zeno_malloc(14);
14   zeno_strcpy(private, "secretpassword");
15
16   int offset = private-public;
17
18   zeno_printf("Printing characters of public array\n");
19   for(int i=0;i<6;i++) {
20     zeno_printf("%c", public[i]);
21   }
22   zeno_printf("\n");
23
24   zeno_printf("Printing characters of private array from public
      array\n");
25   for(int i=0;i<14;i++) {
26     zeno_printf("%c", public[i+offset]);
27     if(public[i+offset] == private[i])
28       test_status = 0;
29   }
30   zeno_printf("\n");
31
32   if(test_status == 0)
33      zeno_printf("Test Failed: OOB Read\n\n");
34
35   capability_exit();
36   return 0;
37 }
```

Listing 2: Capability-Aware Out-of-Bounds Read Example.

The following code snippets step through the example application from Listing 2 to demonstrate capability usage with BGAS-Zeno capability library functions that automatically generate capabilities.

```
1 #include "cap_user.h"
```

Include the `cap_user.h` header file in each application to define the capability library function prototypes described in Table 1.

```
3  int main() {
4    capability_init();
```

Just as in Example 1, `capability_init()` must be called before using capability features.

```
6    zeno_printf("\nStarting test: OOB Read\n");
```

Capability-aware implementations and wrappers of standard C functions are prefixed with "`zeno_`", but otherwise use the same arguments. Reference Table 1 for a complete list of supported functions.

```
10   char *public      = (char *)zeno_malloc(6);
```

The `zeno_malloc()` function automatically creates a capability of the requested size and associates it with the returned pointer. The Zeno-llvm compiler ensures the capability ID is used for all memory accesses with the new pointer.

```
25   for(int i=0;i<14;i++) {
26     zeno_printf("%c", public[i+offset]);
27     if(public[i+offset] == private[i])
28       test_status = 0;
29   }
```

The `for` loop in lines 25-29 creates the out-of-bounds read violation. The `public` pointer is indexed beyond the original six bytes allocated with `zeno_malloc()`, triggering a capability access violation which terminates the application.

```
35   capability_exit();
36   return 0;
37 }
```

As in Example 1, `capability_exit()` must be called before returning from `main()`.

Executing the example application without capability protections produces the following output. Note that the secret string is successfully printed using the `public` pointer that was not intended to reference that data.

```
/root/benchmarks/riscv # ./example_app2-riscv.exe

Starting test: OOB Read
Printing characters of public array
public
Printing characters of private array from public array
secretpassword
Test Failed: OOB Read

/root/benchmarks/riscv #
```

Executing the example application with capability protections produces the following output. When attempting to print the secret data with the `public` capability, a capability fault is raised by the hardware and the application is terminated. Capability print statements generated by firmware may appear out-of-order with respect to strings printed with `zeno_printf()` because of buffering performed by the operating system.

```
/root/benchmarks/riscv # ./example_app2-zeno.exe

Starting test: OOB Read

[C_CREATE] C: 0x0000000000000006
    Min:  0x00000000000152A0
    Max:  0x00000000000152A6
    Perm: 0x0000000000000007

[C_CREATE] C: 0x0000000000000007
    Min:  0x00000000000152C0
    Max:  0x00000000000152CE
    Perm: 0x0000000000000007
Printing characters of public array
public
Printing characters of private array from public array
[ 1094.330829] example_app2-ze[58]: unhandled signal 4 code 0x4 at 0x0000000000010764 in example_app2-zeno.exe[100]
[ 1094.333989] CPU: 0 PID: 58 Comm: example_app2-ze Not tainted 5.19.0-dirty #21
[ 1094.334692] Hardware name: Secure Micro Technologies - BGAS-Zeno (DT)
[ 1094.335164] epc : 0000000000010764 ra : 000000000001071c sp : 0000003fc22f5b90
[ 1094.335747]  gp : 0000000000014800 tp : 0000003f83ffa380 t0 : 0000003f84023290
[ 1094.336309]  t1 : 000000000001049c t2 : 0000000000000003 s0 : 0000000000000000
[ 1094.337342]  s1 : 0000000000000001 a0 : 00000000000152c0 a1 : 00000000000152a0
[ 1094.337975]  a2 : 0000000000000001 a3 : 0000000000000021 a4 : 0000000000000000
[ 1094.338524]  a5 : 0000000000000000 a6 : 0000000000000004 a7 : 0000000000000040
[ 1094.339076]  s2 : 000000000000000d s3 : 0000000000013e18 s4 : 0000000000010574
[ 1094.339628]  s5 : 0000003fc22f5d68 s6 : 0000000000013e18 s7 : 0000003f84021d78
[ 1094.340195]  s8 : 0000003f84022030 s9 : 0000000000000000 s10: 0000000000000000
[ 1094.341213]  s11: 00000000002347b8 t3 : 0000003f83f0e38c t4 : 000000000009338c
[ 1094.341869]  t5 : 0000000000023018 t6 : 000000000034e6b4
[ 1094.342347] status: 0000000200000020 badaddr: 0000000000000006 cause: 0000000000000015
Illegal instruction
/root/benchmarks/riscv #
```

The dump info provided by Linux includes the faulting program counter value (`epc`), the capability ID that caused the fault `badaddr`, and the type of fault (`cause`). See the BGAS-Zeno ISA specification for more details about the `cause` codes.

4 Zeno Capability Model

1. Capability Definition

In the BGAS-Zeno architecture, every memory operation is a capability-based memory access. Capabilities attach metadata to each memory operation. Metadata stores access permissions associated with the given memory reference. Hardware enforces the access permissions, preventing software vulnerabilities related to memory safety.

The BGAS-Zeno architecture uses an extended addressing model to encode a Capability ID in each memory reference. Extended memory references are formed from two separate 64-bit components, a memory pointer and capability ID. The memory pointer portion of the reference represents a memory address. The capability ID component of the address serves as an access token, referencing access permissions in metadata. Figure 8 depicts a complete memory reference in the BGAS-Zeno architecture.

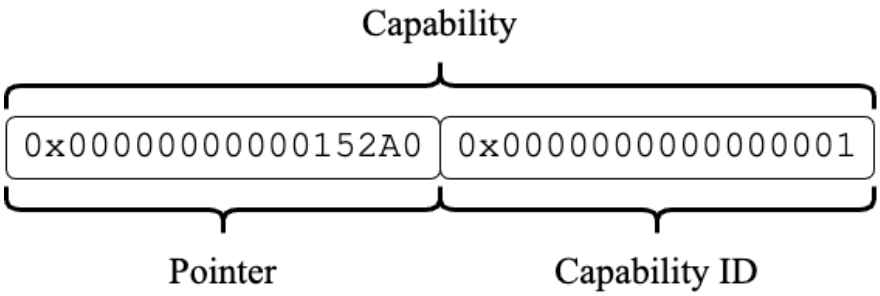


Figure 8: BGAS-Zeno Capability-based memory reference.

Each capability provides read, write, or execute access permission to a range of addresses. Any software context can request the creation of a new isolated capability from hardware. In BGAS-Zeno, capabilities are hierarchical, allowing a new capability with monotonically decreasing access permissions to be derived from an existing capability. Access permissions provided by a capability may be revoked, preventing further data access with the revoked capability or any capabilities derived from it.

2. Capability Operations

The BGAS-Zeno architecture provides three capability manipulation operations, in addition to the instructions described in the BGAS-Zeno ISA. Future hardware implementations of these operations will refine their functionality. Therefore, to ensure compatibility with future versions, it is recommended to use the API operations described in Section 3 to invoke these operations instead of directly executing the dedicated hardware instructions.

- **Create Capability -** Creates a new capability to serve as an isolated container of memory with user requested access permissions and min/max address ranges. The new capability ID is written to the capability register file for use by the application.

- **Derive Capability -** Creates a new capability with permissions less than or equal to the parent capability it was derived from. Permissions are requested by the user and validated in hardware. The new capability ID is written to the capability register file for use by the application.
- **Revoke Capability -** Deletes access permissions associated with a capability and all of the capabilities derived from it. However, the associated capability IDs in memory and on-chip are not invalidated. Future memory accesses may be attempted with these stale capability IDs, causing access faults because of invalid permissions.

4.3 Capability Metadata Format

Capability Metadata is only directly accessible to hardware and trusted firmware. The BGAS-Zeno architecture separates Metadata into fixed size and variable size components. Each capability requires 64 Bytes of fixed-size Metadata, divided into 8 Byte Fields. The BGAS-Zeno Metadata format is described by Table 2.

Table 2: BGAS-Zeno Metadata format.

Offset	Field Name	Description
0x00	Minimum Address	The minimum address accessible to the capability.
0x08	Maximum Address	The maximum address accessible to the capability.
0x10	Permissions	64-bits of access permissions for the given address range. Bits 2, 1, and 0 represent read, write, and execute respectively. Bits 63-3 are reserved for future use.
0x18	Translation Info	Reserved for future per-capability address translation support. Future versions will use this field to reference an address translation page table.
0x20	Root Capability ID	The capability ID of the capability at the root of the capability hierarchy the Metadata is a part of.
0x28	Parent Capability ID	The capability ID of the capability that derived the capability the Metadata is associated with.
0x30	Children Pointer	A memory pointer to a list of capability IDs derived from the associated capability.
0x38	Reserved	Reserved for future use.

Currently, variable sized Metadata stores only the list of children capabilities. Future versions of BGAS-Zeno will extend support to include space for virtual address translation page tables and potentially other variable sized data structures.

What is the Box?

- BGAS-Zeno Development Board
- 12V Power Supply
- USB Mini-B Cable