Secure Input Validation in Rust with Parsing-Expression Grammars

by Madeleine Dease Dawson

S.B., C.S. M.I.T., 2017

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology
February 2019

Author .....
Department of Electrical Engineering and Computer Science
Feb 1, 2019

Certified by .....
Howard E. Shrobe
Principal Research Scientist, MIT CSAIL
Thesis Supervisor

Certified by .....
Hamed Okhravi
Senior Staff Member, MIT Lincoln Laboratory
Thesis Supervisor

Certified by .....
Richard W. Skowyra
Technical Staff Member, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Secure Input Validation in Rust with Parsing-Expression Grammars

by

Madeleine Dease Dawson

Submitted to the Department of Electrical Engineering and Computer Science
on Feb 1, 2019, in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Accepting input from the outside world is one of the most dangerous things a system can do. Since type information is lost across system boundaries, systems must perform type-specific input handling routines to recover this information. Adversaries can carefully craft input data to exploit any bugs or vulnerabilities in these routines, thereby causing dangerous memory errors.

Including input validation routines in kernels is especially risky. Sensitive memory contents and powerful privileges make kernels a preferred target of attackers. Furthermore, the fact that kernels must process user input, network data, as well as input from a wide array of peripheral devices means that including such input validation schemes is unavoidable.

In this thesis we present Automatic Validation of Input Data (**AVID**), which helps solve the issue of input validation within kernels by automatically generating parser implementations for developer-defined structs. AVID leverages not only the unambiguity guarantees of parsing expression grammars but also the type safety guarantees of Rust. We show how AVID can be used to resolve a manufactured vulnerability in Tock, an operating system written in Rust for embedded systems.

Using Rust's procedural macro system, AVID generates parser implementations at compile time based on existing Rust struct definitions. AVID exposes a simple and convenient parser API that is able to validate input and then instantiate structs from the validated input. AVID's simple interface makes it easy for developers to use and to integrate with existing codebases.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist, MIT CSAIL

Thesis Supervisor: Hamed Okhravi
Title: Senior Staff Member, MIT Lincoln Laboratory

Thesis Supervisor: Richard W. Skowyra
Title: Technical Staff Member, MIT Lincoln Laboratory

Acknowledgments

I would like to extend my sincerest thanks to Drs. Hamed Okhravi and Howie Shrobe for welcoming me into the Inherently Secure group and giving me the opportunity to work on an exciting and engaging project. I am appreciative of not only the guidance they provided me related to my research and writing but also their educative insights into the realm of security.

I am indebted to my advisor Dr. Rick Skowyra for spending countless hours advising my research and providing feedback on my writing. I am especially appreciative of his mentorship regarding the field of formal language theory, a topic with which I was previously not well acquainted. His patience with and confidence in me provided me with the motivation to persist through the challenges I faced.

I would also like to thank the rest of the Inherently Secure group, including both current and former students. Their camaraderie and debugging assistance helped make the difficulties of embedded systems programming more bearable.

My parents, John and Ann, have been nothing but supportive throughout my entire education, and for that I must offer my deepest thanks. They taught me the value of hard work and determination, which undoubtedly has gotten me to where I am today.

Finally, I would like to offer heartfelt thanks to my friends, family, and cats for providing me with emotional support as well as tolerating my long and sometimes odd hours. Their patience and kindness is invaluable to me.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In the age of cyberwarfare, mission-critical systems depend on security for correct operation. One facet of security is that of memory safety: the state of a system's memory being safe from security vulnerabilities. For systems with sensitive memory contents, input-output interfaces are especially popular channels of attack. Attackers can provide data of their choosing, often specifically crafted to exploit bugs that compromise memory safety. The ability to handle such input data while maintaining confidentiality, integrity, and availability is difficult, especially for systems with complex, monolithic kernels and embedded systems with many different peripheral devices.

Processing user or device input often involves reading data from memory into structures, or structs. Structs can contain multiple fields with different types, and developers often manually write code to verify that given data can be interpreted as a certain struct. Algorithms that execute such procedures are prone to developer-introduced bugs and logic errors; attackers can then craft input to exploit these specific errors.

Errors of this type are classified by the Common Weakness Enumeration (CWE) as "Improper Input Validation" [1]. Classes of vulnerabilities within this category include buffer overflow and type confusion, which happens when data of one type is interpreted as data of another type. These vulnerabilities can be exploited to produce denial of service and code execution attacks. For example, in July 2018,

a vulnerability in Adobe Flash Player was disclosed where user-specified data could cause a type confusion error, leading the program to execute code included in the attacker's payload [2] [3].

Errors such as these are especially dangerous in operating system kernels. In a system that performs critical tasks or stores sensitive data (e.g., credit card information), an adversary who is able to execute arbitrary code could do immeasurable damage. Therefore, a common objective of modern operating system engineering is to ensure memory safety, especially when interpreting user input.

One operating system that emphasizes memory safety is Tock. Tock is written in Rust, a type-safe language that uses novel mechanisms to prevent spatial and temporal memory errors. Another security-mindful Tock design choice is its kernel architecture, which resembles a microkernel. Microkernels compartmentalize services so that each component requires minimal permissions—when one of these components is compromised, the amount of harm an adversary can do to the system is limited.

However, even in a supposedly safe operating system, structs still need to be explicitly cast from memory. For example, when reading process metadata from flash memory, Tock must perform manual checks on the input data. Any error in performing this cast could, either directly or via the created process, violate the memory safety of the kernel. For this reason, we see Tock as a promising target for the work presented below, which extends the Rust language with mechanisms to reduce or eliminate such input handling errors.

Our work enables bug-prone manual input validation to be replaced with automatically generated input parsers. All input is parsed unambiguously using a provably sound parsing algorithm generated at compile-time. To do so, this work leverages parsing expression grammars (PEGs) to automatically validate input before it is stored in user-specified structures. Per-struct PEGs are derived automatically from struct definitions, and the associated parser can be invoked by the developer on any untrusted data before it is cast into a struct.

Since PEGs are unambiguous, attempting to parse user data into a struct either succeeds as intended or fails without compromising the system. Together, Rust's

type-safety guarantees and PEG parsers' unambiguity guarantees provide the system with a secure mechanism for I/O handling. Integrating this parsing mechanism with Tock removes the need to manually cast structs in the kernel, thereby eliminating the possibility of dangerous input mishandling bugs. While this thesis uses Tock as an application for the parser generator, the system can also be used as a stand-alone tool for any Rust project.

1.1 Objective and Goals

The objective of this work is to replace bug-prone manual input validation with automatically generated input parsers that implement a provably sound validation algorithm. Furthermore, we would like this work to help developers use the minimum amount of "`unsafe`" Rust as possible. `Unsafe` Rust code can perform certain actions on memory that safe Rust cannot, such as dereferencing arbitrary raw pointers. Using `unsafe` Rust voids some of the safety guarantees provided by Rust; since we want to rely on these guarantees for the safety of our parser, we would like to avoid using `unsafe` Rust.

Because this work is motivated by and in support of Tock, the parser generator is specifically intended for the types of data structures used by the kernel. These structs are often relatively simple, containing only a few fields which are either Rust primitives (e.g., integers, booleans) or references to other kernel data structures. More complicated recursive data structures, such as a graph node struct a user might implement as part of a graph algorithm, are out of scope for this system.

This work emphasizes system usability by prioritizing functionality, efficiency, and minimal developer effort. The system should be able to support Rust structs whose fields are an arbitrary combination of primitives and references to previously-defined structs. Furthermore, the system should perform parsing tasks with minimal temporal and spatial overhead. Finally, the system should require minimal extra effort from developers. Developers should be able to define Rust structs normally and simply add an annotation to invoke parser generation and execution.

1.2 Contributions

This thesis designs, implements, and validates a type-safe parser generator-based parsing mechanism, which is invoked with a simple source code annotation, and shows that it prevents certain classes of vulnerabilities and attacks. Chapter 2 introduces common attacks on input handlers. Chapter 3 describes relevant theory behind formal languages, including the differences between context-free grammars and parsing expression grammars, two types of formal languages often employed by input parsers. Chapter 4 introduces and motivates our system. Chapter 5 describes the implementation in detail as well as its integration into Tock. Chapter 6 discusses classes of attacks prevented by the parsing system. Chapter 7 measures the performance of the system after including the parsing modifications. Chapter 8 compares the system to other parser generators and previous memory safety approaches. Chapter 9 concludes and outlines future work.

Chapter 2

Memory Safety and Common Vulnerabilities

Within a computer system, a vulnerability is a weakness created by a logic error or bug such that an adversary can take advantage of this weakness to gain some sort of control over the system. An exploit is the specific procedure the adversary executes to trigger the vulnerability. Oftentimes the exploit involves the adversary directly providing the system with data, or input. Since the adversary has complete control over the input, they can carefully craft it to trigger any vulnerabilities within the system. Bugs or mistakes in input-handling code are dangerous because they threaten memory safety and therefore invite memory corruption attacks.

Essential to the security of a system is memory safety. Ensuring the memory safety of a system requires protecting it from memory errors, that is, vulnerabilities that expose the potential for memory corruption. Memory errors can be divided into two categories: spatial and temporal [4]. Spatial errors occur when an out-of-bounds pointer is dereferenced; temporal errors occur when a pointer to an invalid or freed value is dereferenced. These errors, possibly combined with an adversary's input, can be used to carry out a range of attacks from simply leaking memory to controlling the flow of the program.

To better understand memory errors, we must first review the regions of memory used during code execution. Random access memory (RAM), also called main mem-

ory, is a form of primary memory used in computing systems. When storing data, RAM is favored over mass storage devices such as disks and solid state drives due to its low read and write latency from the central processing unit (CPU). In most systems, RAM is divided into four main regions: the stack, the heap, text, and data. The latter two regions store code and static (global) variables, respectively.

The stack stores variables related to function calls during runtime, and the heap is primarily used to dynamically allocate blocks of memory for objects. During each function call, a stack frame comes into scope. The stack frame contains data related to the function call itself, such as parameter values and local variables, as well as information on what to do after the function call completes. The latter is accomplished by saving a return address that points to the next instruction to execute after the function returns. For example, if function a calls function b, function b's stack frame will contain a return address that points to the instruction directly after function b is called within function a.

Because the saved return address specifies an instruction to execute, it is often the target of memory corruption attacks. Since local variables live on the stack near the current return address, an adversary might try to exploit a spatial error by corrupting the return address via a local variable. Executing this exploit with carefully crafted input allows the adversary to set the return address to an address of their choosing. They can thereby control the flow of the program and, depending on the protections in place or lack thereof, execute arbitrary code.

An attacker can still corrupt program control flow without touching the return address or going out-of-bounds of an object. For example, input that triggers a divide-by-zero error may crash a system; this is a denial-of-service attack. Another type of memory error is a type confusion error, which happens then memory is accessed as an unintended or incompatible type [5]. This memory error can again be exploited by an adversary so that the system performs some otherwise prohibited actions. For example, if a casting algorithm accidentally interprets a region of memory containing user input as a struct that contains a field that is a pointer or reference, an adversary could craft their input to set this pointer to point to an address of their choosing.

2.1 Example: Parsing a Struct

Input from users or peripheral devices is often unstructured, so developers must write code that explicitly performs a cast for the given data type. The omission of important input checks can create vulnerabilities that may not be obvious during code review or testing. Complex corner cases lead to bugs that are rare and hard to identify; these bugs can cause vulnerabilities. Writing code that handles user input leaves little room for error.

Consider the following example, where a developer writes a function to cast some integers into a simple struct. `MyStruct`, whose definition is shown in Listing 2.1, has three fields, all integers, or `u32`'s in Rust notation—unsigned 32-bit integers. The parsing function, `parse_struct`, takes in a fixed-size array of three `u32`'s—one for each field of `MyStruct`—and then returns an instance of `MyStruct` with the fields set. This function is shown in Listing 2.2. Line 2 creates an empty instance of `MyStruct` called `my_struct`; line 3 creates a raw pointer to the instance; lines 5 through 7, within the `unsafe` block, supposedly fill in the instance with the provided integers; line 9 returns `my_struct`.

The developer, knowing that the stack grows downward, may deduce that decreasing the value of the pointer to `my_struct` will give them access to subsequent fields of the struct. They decide to use Rust's `sub` method to subtract an offset from the pointer [6]. However, perhaps confusingly, this method does the opposite of what the developer wants and instead increases the value of the pointer by a certain offset. While the first member of `fields` (the input) will end up in `field_1` of `my_struct`, `parse_struct` will end up overwriting existing stack data with the second and third members of the input array. Specifically, `parse_struct` will end up overwriting the return address, thus changing the execution of the program. Figure 2-1 shows the stack frame corresponding to this function call.

Even those unfamiliar with Rust will see that this example is particularly contrived; it requires that the developer be unaware of not only the Rust documentation but also the fact that the input can be passed into the `MyStruct` constructor. Nev-

ertheless, it is not uncommon for developers to publish buggy, untested code. This example also shows that the usage of Rust's `unsafe` keyword, necessary for raw pointer arithmetic and dereferencing, is dangerous—`unsafe` code cannot protect the developer from spatial and temporal errors. In an ideal world, `unsafe` code would be consolidated into trusted and vetted libraries and only accessible through provably safe APIs. This work contributes to this effort by ensuring that developers do not have to manually cast or parse structs.

```
1  struct MyStruct {
2      field_1: u32,
3      field_2: u32,
4      field_3: u32,
5  }
```

Listing 2.1: A simple Rust struct with three fields, all `u32` integers.

```
1  fn parse_struct(fields: [u32; 3]) -> MyStruct {
2      let mut my_struct: MyStruct =
3          MyStruct { field_1: 0, field_2: 0, field_3: 0 };
4      let raw_pointer = &mut my_struct as *mut MyStruct;
5      unsafe {
6          *(raw_pointer as *mut u32) = fields[0];
7          *(raw_pointer as *mut u32).sub(1) = fields[1];
8          *(raw_pointer as *mut u32).sub(2) = fields[2];
9      }
10     my_struct
11 }
```

Listing 2.2: A function that parses three `u32` integers into an instance of `MyStruct`.

Beside spatial errors, buggy code can also create type confusion errors. One issue that may invite a type confusion error is that of endianness. The concept of endianness deals with the order in which bytes appear in memory. In a big-endian representation, the first or most significant byte appears first (i.e., at the lowest address) in memory; in a little-endian representation, the least significant byte appears first. Accessing bytes in the wrong order as intended is a type confusion error.

EBP+16	fields
EBP+12	
EBP+8	
EBP+4	Return address
EBP	Saved base pointer
EBP−4	field_1
EBP−8	field_2
EBP−12	field_3

Figure 2-1: The stack frame for the `parse_struct` function call.

For an example of what might go wrong, consider a program that calculates the slope of a line. It does so given a hard-coded point, $(1, 1)$, and a user-provided point, represented as two 32-bit integers. The program was first written on a big-endian machine and accessed the bytes of the integers in the correct order. The developer then loaded the code on a little-endian machine, at the same time remembering to add a check to make sure the user's provided y-coordinate does not equal 1, thus preventing a critical divide-by-zero error. While this check accesses the y-coordinate in the now correct little-endian order, the developer forgets to update the slope calculation code. Therefore, the panic-inducing divide-by-zero error is not actually prevented; it can be exploited by an adversary to perform a denial-of-service attack.

This example illustrates that input handling can be architecture-specific and that failing to account for this causes memory errors. Another architecture-specific detail is word size, which determines pointer size and therefore the size of types that represent pointers. We wish to hide these details from developers who may not understand or account for them. Since our parser generator system requires knowledge of these details, we would like the system to be able to accept and then remember the provided options. This way, the parser generator can be configured once, and developers who invoke the parser do not have to consider the architecture on which their code is running.

Chapter 3

Formal Grammars

Since our solution to the I/O safety problem employs parser generators, we must first provide some background on formal language theory and grammars. In general, the goal of formal language theory is to study language syntax, or constraints on the structure of sentences in a given language. A formal language is defined by an alphabet, or a set of symbols, and certain rules that determine which combinations of symbols are valid. For a given language, these rules can be described by a formal grammar. A grammar is used to build the set of strings that are valid under the language it describes.

According to Noam Chomsky's 1956 article "Three Models for the Description of Language", a formal grammar, which is comprised of symbols and production rules, can be defined by the tuple $G = N, T, P$ [7]. Symbols can be either terminal or nonterminal. The set N contains nonterminal symbols, which are essentially variables used only during string production; they do not appear in any strings produced by the grammar. N also contains the start symbol S, which is used to begin string production. The set T contains terminal symbols; these are the symbols that appear in strings generated by the grammar. T is disjoint from N. The set P contains production rules, which are the actions performed on a string to transform it. Rules take the form $(T \cup N)^* N (T \cup N)^* \rightarrow (T \cup N)^*$—that is, each rule maps a combination of terminal symbols with at least one nonterminal symbol to a combination of terminal and nonterminal symbols.

3.1 Parser Generators

Although first proposed by Chomsky for use in the study of natural language, formal grammars are now important tools in computing applications such as specifying programming language syntax. As part of their specification, programming languages define a grammar that describes valid sequences of tokens. When compiling code, the compiler must validate that the sequence of tokens can be generated by the grammar. To validate such input is to parse it; an object that performs this action is called a parser. The sequence of production rules that built the string is called the derivation of the string.

There are a few properties that we would like a parser to have. First, it should be safe—that is, free of memory errors. As discussed in the previous section, bugs in input-handling code can create dangerous vulnerabilities that adversaries can carefully exploit using malicious input. Second, the parser should behave deterministically. Given input to be parsed, there should be at most one possible derivation, and the result of the parse should reflect this derivation. This property is important because it is difficult to reason about nondeterministic behavior. Sometimes interpreting input as one type and sometimes as another type invites the possibility of type confusion errors, which we want to avoid.

Finally, we wish our parser to be efficient. Different parsing algorithms have different space and time complexities. We would like the parser to have spatial overhead viable on lightweight microcontrollers, the type of machine on which Tock runs. Furthermore, since we intend for struct parsing to happen in the kernel, we would like an efficient parsing algorithm that does not add significant performance overhead to the operating system.

3.2 Context-Free Grammars and Parsing Expression Grammars

Beside introducing this formalization, "Three Models for the Description of Language" also introduced what is now known as the "Chomsky hierarchy" of formal grammars. This hierarchy categorizes grammars by the forms of production rules allowed. Type-0 grammars, or unrestricted grammars, allow any production rules that follow the pattern stated above. Type-1, -2, and -3 grammars permit increasingly strict subsets of production rules. For example, type-2 grammars, or context-free grammars (CFGs), only include production rules that map a single nonterminal symbol to a combination of terminal and nonterminal symbols (i.e., $N \rightarrow (T \cup N)^*$).

Since first introduced by Chomsky, CFGs have been a popular choice for programming language grammars. The first language to use a CFG to specify syntax was ALGOL [8]; more recent uses of CFGs in programming languages include the Extensible Markup Language (XML) [9] and Google's Go [10]. GNU Bison, widely used for creating programming language parsers, generates a parser for a language based on the language's CFG [11].

Among programming language grammars, CFGs are popular because they fall in a sweet spot along the Chomsky hierarchy. Type-3 grammars, or context-sensitive grammars, are more restrictive than CFGs and require parsing algorithms that keep a significant amount of state. Because this extra overhead can be expensive, CFGs are more desirable than context-sensitive grammars. On the other hand, less restrictive grammars pose their own problems. It has been shown that unrestricted grammars are equivalent to Turing machines—deciding whether a string was generated by an unrestricted grammar is equivalent to the Halting problem, which is undecidable [12].

One downside of CFGs is that a single string can have multiple derivations. For example, consider the following grammar.

$$
\begin{aligned}
T &= a \\
N &= S \\
S &\rightarrow S + S \mid S * S \mid a
\end{aligned}
\tag{3.1}
$$

This grammar describes a simple arithmetic expression that uses only addition and multiplication. It consists of a single terminal symbol, a; a single nonterminal symbol, S, or the start symbol; and a single production rule. This rule provides three choices for replacing S: addition of two S, multiplication of two S, or the terminal a. The order in which these operations are applied determines the value of the expression.

Consider the string $a * a + a$ produced by this grammar. This string has two derivations: the first derivation consists of the first rule option (addition) followed by the second rule option (multiplication), and the second derivation consists of the second rule option followed by the first rule option. This example shows that CFGs can indeed have multiple derivations and can therefore be ambiguous. This ambiguity leads to nondeterminism, which in turn leads to type confusion errors.

However, our problem statement necessitates that we interpret struct data without ambiguity in order to avoid such type confusion errors. We therefore choose instead to work with a different class of grammars, called parsing expression grammars (PEGs). Similar to CFGs, PEGs consist of a set of terminal symbols, a set of nonterminal symbols, and a set of production rules [13]. These production rules take the form $N \rightarrow e$, where N is a nonterminal and e is an expression, again requiring that the left-hand side consist of only a single nonterminal.

The main difference between PEGs and CFGs is in how PEGs handle expressions. The right-hand side of a CFG production rule can be the empty string, a single terminal, a single nonterminal, or a sequence of these symbols. It is also possible to specify a repetition of symbols using the Kleene star. PEG expressions consist of all of these options as well as a prioritized choice operator, which lists multiple expressions eligible for replacement in priority order.

This prioritized choice operator is specific to PEGs and plays a key role in the unambiguity of PEGs. When parsing a string generated by a PEG, the prioritized choice operator enforces that even if multiple expressions match the operator, only one is selected, and it is selected deterministically based on the order of expressions specified in the grammar. Whereas multiple derivations might exist when parsing a string relative to a CFG, the equivalent PEG would contain a prioritized choice operator that effectively decides which derivation is chosen during parsing.

Consider again the grammar shown in (3.1). Common sense tells us that when presented with the expression $a * a + a$, we apply the arithmetic order of operations to perform the multiplication first. This correct ordering corresponds to the first derivation described above, where we applied the addition rule option before the multiplication option. If we convert the CFG to a PEG, we can use the prioritized choice operator in the rule to enforce the order of operations. The resulting PEG will correctly ensure that multiplication is always performed before addition.

One further distinction between PEGs and CFGs is that the PEG Kleene star is greedy. When parsing, the expression within the repetition is consumed as many times as possible. This behavior removes the possibility for ambiguity when deciding whether an expression appears n or n+1 times. While a CFG parser would explore both possibilities, a PEG parser simply chooses the greedier option (i.e., the $n + 1$ option).

Because of the unambiguity of PEGs, PEG parsers are more efficient than CFG parsers. While CFG parsers require $O(n^3)$ time to produce a derivation, PEG parsers require only $O(n)$ time with $O(n)$ memoization. For a system that makes use of PEG parsers to cast critical structs at startup, this performance speedup is crucial. Together, PEGs' unambiguity and efficiency should make it clear that PEGs are a good choice for our parser generator system.

Chapter 4

AVID: Solution Overview

To solve the critical issue of improper input handling and validation, we have developed a system that can automatically generate parsers for developer-defined structs. Automatic Validation of Input Data (AVID) is motivated by Tock, partially due to a potential vulnerability that we discovered in its kernel. Tock, an operating system written in Rust, is designed to run on lightweight microcontrollers [14]. It is inspired by microkernel architectures; the kernel is divided into three sections: the core kernel, platform-specific device drivers, and kernel extensions called capsules.

Unfortunately, the low-level operations required of an operating system often involve manipulating raw pointers, necessitating `unsafe` Rust code. For example, register addresses vary by board and therefore must be hard-coded. These hard-coded addresses are then accessed via raw pointers. In Tock, this `unsafe` code exists solely in the core kernel and device drivers, where it is absolutely necessary. Because these components use `unsafe` code, they must be trusted to have been implemented correctly and cannot be verified by the compiler.

Trusting code is a famously bad idea [15], and the Tock architecture minimizes the amount of trusted code by requiring that capsules, which are untrusted, not use `unsafe` code. Capsules implement key features such as encryption logic, driver interfaces, and timers. When a developer wants to add new capabilities to Tock, they implement new capsules to run on top of the core kernel without writing `unsafe` code. This way, the trusted codebase rarely changes or needs to be re-verified.

Theoretically, at least, adding arbitrary safe Rust code to Tock capsules should not compromise the security of the operating system. However, the potential vulnerability we discovered contradicts this idea. Tock's console capsule, which provides an interface for processes to write to peripheral devices, is used by Tock's implementation of the `print` function [16]. Specifically, the implementation of `print` reserves a buffer of size 64 within kernel memory and then passes this buffer, along with the data the user desires to print, to the console capsule.

Unfortunately, the capsule relies on manual, developer-inserted checks to enforce that the user data fits within the allocated buffer. To confirm this, we removed the checks and provided improperly sized input from a user application. The result was a spatial error that enabled a buffer over-read attack. As a result, the `print` function ended up printing kernel memory outside of the allocated buffer. This information leak vulnerability, dubbed "Information Exposure" under the CWE, can expose sensitive data that would be of interest to an adversary (e.g., encryption keys) [17].

Although this capsule vulnerability is merely a demonstration of a potential developer error, it illustrates the need for a safe input-handling mechanism within the Tock kernel. Furthermore, it shows that even supposedly safe Rust can, if buggy, expose vulnerabilities in the operating system. Therefore, it is not enough to simply minimize the amount of `unsafe` code; we must also take extra measures to prevent input mishandling in the kernel. It should be clear that Tock would indeed benefit from a parser generator system that validates input.

AVID secures struct parsing by automatically generating parser implementations for structs and allowing the developer to invoke them with a single line of code. The system harnesses the power of Parsing Expression Grammars (PEGs), which, as previously described, are unambiguous and therefore allow for efficient parsing. Requiring minimal developer effort to integrate with existing codebases, the system protects against memory errors that may arise when manually casting structs. Furthermore, it absolves developers of the need to think carefully about type casting and architectural details.

We use Pest, an open-source PEG parser generator written in Rust, to create

struct parsers at compile time [18]. Pest takes advantage of Rust's procedural macro system to generate parser implementations based on the tokens being compiled. While Pest is designed for parsing ASCII characters based on a grammar defined in a separate text file, we want to impose as little additional work on developers as possible, and enable Rust struct definitions to be automatically translated into grammars. Therefore, significant modifications were made so that Pest's parsing algorithm could be applied to struct definitions.

The contributions of this work include a **compiler macro**, using Pest's PEG parsing algorithm, which can generate parser implementations from Rust struct definitions. Developers mark structs for parser generation using a **novel derive attribute**, shown in the first line of Listing 4.1. This attribute tells the Rust compiler at compile time to invoke the macro described above on the annotated struct definition. Architectural details are configured via **AVID-specific environment variables**, which can be configured in a central location so that such details are mostly abstracted away.

Our **new, compiler-generated parser** exposes a simple and convenient API that is easy for developers to invoke. Listing 4.2 shows an example usage of a generated parser. The developer has some input data, represented by the `input` variable as an array of bytes. To interpret these bytes as an instance of `MyStruct`, the developer simply invokes the method `parse_and_create`, which first verifies that the sequence of bytes is valid under the struct grammar and subsequently casts the bytes to an instance of `MyStruct` and returns the object.

```
1 #[derive(Parser)]
2 struct MyStruct {
3     f1: u16,
4     f2: u16,
5     f3: u16,
6 }
```

Listing 4.1: Annotated Rust struct definition.

```
1 input = [...];
2 let ms: MyStruct = MyStruct::parse_and_create(input);
```

Listing 4.2: Parser invocation.

Motivated by the potential Tock vulnerability example we found, we have integrated AVID with the Tock kernel. Even through Tock's core kernel is trusted, it is still important to reduce and consolidate the amount of `unsafe` code used; AVID helps accomplish this goal. Following the integration, we show that certain key structs critical to Tock can be safely cast from memory. This application demonstrates that it is feasible and worthwhile to invest in input-handling safety mechanisms, which protect systems against critical I/O-related memory errors.

Chapter 5

AVID Implementation

We now discuss the implementation of AVID, our input verification solution for kernel-based I/O. Intended to replace possibly buggy and therefore insecure input handling in sensitive codebases, AVID absolves developers of the need to manually write struct parsing routines. AVID currently supports the parsing of simple structs containing Rust primitives, as typical of kernel data structures. Integrating AVID into existing systems requires minimal developer effort while preventing the types of careless errors previously presented, such as certain spatial and type confusion errors.

AVID replaces manual struct parsing code by automatically generating parser implementations based on struct definitions. To tag a struct for parser generation, which happens at compile time, developers simply include a single-line source code annotation above the struct. Using the simple parsing API that AVID exposes, developers can then invoke the generated parser to validate input and instantiate a struct based on this input. Critical architectural information required for correct parsing and instantiation, such as endianness and word size, is configured in a central location and abstracted away from developers. This design emphasizes usability while also minimizing the effort needed to integrate AVID into existing codebases.

AVID works by leveraging both an existing PEG parser generator implementation, called Pest, as well as built-in Rust tools. We chose to work with Pest because it not only provides us with a working implementation of the packrat parsing algorithm used for efficient PEG parsing but also leverages Rust's procedural macro tools

to generate parsers [19]. Unlike standard Rust macros, which many other parsing libraries use, procedural macros take as input the code being compiled—this is useful for our purposes since we would like AVID to generate parsers based on only existing struct definitions.

5.1 Extensions to Pest

Pest was originally intended to parse strings of ASCII characters, while AVID parses arrays of raw binary data. Extending Pest to support this required significant modifications. First, we needed to change the mechanism for specifying the grammar. Originally, Pest accepted grammars via text files provided as an additional argument to the procedural macro. This is inconvenient for AVID, as we would like the existing Rust struct definition to act as the grammar. Passing the struct information through a text file would not only be redundant but also invite the possibility of dangerous inconsistencies. Instead, we extended the procedural macro to determine the struct definition from the tokens passed to it at compile time. The resulting interface is simpler, less error-prone, and easier for developers to use.

Unlike AVID, Pest's original use case of parsing ASCII characters required no knowledge of data types. To allow our toolchain to parse and instantiate structs, we augmented Pest with built-in parsing and generation rules for each Rust primitive. These rules are based on preexisting built-in Pest rules, which were previously relevant for ASCII parsing—one example is a WHITESPACE rule that matched a single whitespace character (e.g., space, tab). Each of our new primitive parsing rules consumes from the input the same number of bytes as the size of the primitive; each generation rule casts these bytes to the respective type.

However, knowing how many bytes to consume and in which order to consume them requires knowledge of platform-specific architectural details like word size and endianness. For example, Rust's `usize` and `isize` primitives depend on the architecture's word size. As previously demonstrated in section 2.1, not accounting for these details can cause a careless developer to create dangerous vulnerabilities. To

avoid this issue, we augmented Pest with a simple mechanism to specify such parameters before parser generation. AVID currently supports two such parameters: one for word size and one for endianness. When the primitive rules described above are applied, these parameters, stored in environment variables, are accessed to ensure that bytes are correctly cast.

Another significant change made to Pest is the last step of struct instantiation after a valid parse. While Pest parsers originally returned simply whether an ASCII string was a member of the provided grammar, AVID's use case requires that the validated input be safely cast to an instance of the relevant struct. We achieved this by extending the parser API to provide not only parsing but also instantiation capabilities. Specifically, AVID uses the aforementioned typecasting methods to cast the input to the respective types of each struct field and then returns an instance of the struct with these field values.

5.2 AVID Interface

We now further explore the mechanisms behind the AVID toolchain, illustrated by Figure 5-1. Before compilation, the developer marks each struct intended for parser generation with a Rust derive attribute for AVID's `Parser` trait (1). Within the AVID codebase, the procedural macro entry point is marked as the derive handler for this `Parser` trait. This procedural macro and the tools it uses comprise AVID.

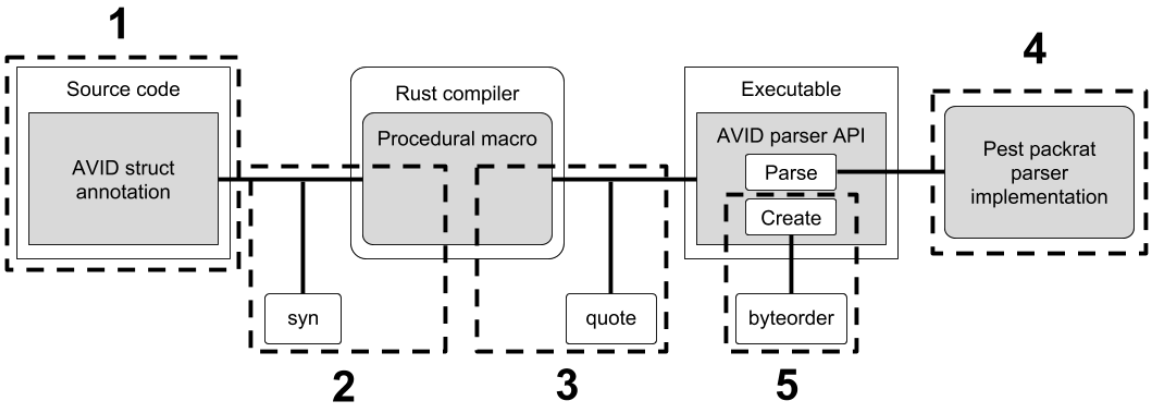

Figure 5-1: AVID toolchain implementation and interface.

During compilation, after the compiler has constructed an abstract syntax tree of the program, it checks for any derive attributes on user-defined types. The compiler isolates the tokens that represent this type, passing them to the respective derive handler in the form of a stream of token trees. AVID then uses `syn` [20], a third-party library, to convert the tokens to types from which the original struct definition information can be more easily accessed (2).

The procedural macro uses this struct information to procedurally generate additional code to add to the program: the parser API, type-casting functions for struct instantiation, and a rule corresponding to the struct grammar. Code generation is made possible by a macro provided by `quote` [21], a third-party library (3). This generated code is returned to the compiler as another stream of token trees, which the compiler then appends to the existing syntax tree of the program.

The aforementioned struct rule is represented as a sequence of rules, each rule corresponding to the respective type of each struct field. AVID outputs this new rule to the compiler as a function that can be accessed whenever the generated parser is invoked. However, this rule method is not meant to be used directly by developers. AVID also outputs an API for interacting with the generated parser, implemented as a trait for the user-defined struct. This API has two main uses: to validate that input bytes can be interpreted as the given struct type, and to instantiate a struct of this type from the input. The functions corresponding to these uses are exposed to the developer as methods of the given struct type and are applied at runtime to provided input data.

AVID's parsing functionality is dependent on the packrat parser implemented in Pest (4). When invoked, the parser uses the outputted struct rule function to match and consume bytes in the order of the struct fields. After a successful parse, AVID's instantiation functionality can be used to return an instance of the given struct type with field values corresponding to the input bytes. Instantiation is made possible by the generation methods mentioned above, which use the third-party library `byteorder` [22] to read bytes as different Rust primitive types (5).

5.3 Design Decisions

There were several key design choices we made while developing AVID that affect its usability, functionality, and interface. One previously mentioned decision is to use Rust struct definitions as the grammars that determine parser behavior. Many other parsing libraries store type grammars in separate files, sometimes represented in a domain-specific language (DSL). Reasons for recording grammars this way include language agnosticism as well as the ability to specify semantics not possible in a programming language's struct definition, such as length fields and magic fields. However, since AVID is specific to Rust and does not support such special fields, duplicating the type information outside of the source code would unnecessarily increase programmer effort and the potential for errors.

When considering AVID's use case, we made a conscious decision to restrict what design patterns the system would support. One popular pattern, especially in networking applications, is a packet type with a length field and a payload of that length. While the presence of the length field may help protect against inadvertent out-of-bounds memory access via the payload, the length field being attacker-controlled still poses security risks for vulnerable systems. For data whose interpretation is influenced by an adversary, there is no way to guarantee that it is safely or correctly parsed. We instead wish to apply AVID to the simpler types of fixed-size structs typically used in kernels.

Another design choice related to AVID's developer interface is the choice to pass to the parsing API a byte slice containing the input to validate. Another option initially considered was to pass a raw pointer to the start of the memory region. However, since raw pointers have no regard for memory regions or object boundaries, reading from a raw pointer without bound could result in spatial errors. We instead choose to have the developer pass in an array of bytes from which and only from which the struct is instantiated. This provides developers with the capability of enforcing memory regions in their application.

Since AVID requires knowledge of architecture-specific details such as endianness

and word size to correctly instantiate structs, we needed a method of specifying these details that was effective as well as usable. In our initial prototype, endianness and word size parameters were passed in to the parser API on each invocation. This pattern is not ideal not only because of information redundancy but also because we would like to hide such details from the average developer.

On a given platform, endianness and word size change very infrequently, if at all. Therefore, these parameters can be hard-coded in a central location and only changed if absolutely necessary. Furthermore, saving these parameters in a central location absolves developers of the need to think about such details when invoking the parser. Therefore, our solution is that before compilation, a maintainer of the given codebase sets certain AVID-specific environment variables that the toolchain later accesses when trying to instantiate structs. These environment variables can be set manually or in a Makefile, if applicable.

5.4 Example: Use Case of AVID

We now show an example use case of AVID. Consider Listing 5.1, which contains the definition for a simple struct that holds system call arguments:

```
1 struct Syscall {
2   driver: u32,
3   command: u16,
4   arg1: usize,
5   arg2: usize,
6 }
```

Listing 5.1: System call struct definition.

To mark this struct for parser generation, we add the derive attribute for the `Parser` trait above the struct definition (shown in Listing 5.2).

```
#[derive(Parser)]
struct Syscall {
  driver: u32,
  command: u16,
  arg1: usize,
  arg2: usize,
}
```

Listing 5.2: Annotated system call struct definition.

During compile time, the tokens of the `Syscall` struct definition will be passed to AVID's procedural macro. Using these tokens, AVID will output a rule to parse a `Syscall` struct, which is a sequence consisting of a `u32`, a `u16`, and two `usize` primitives. However, since the size of a `usize` primitive depends on the word size of the target, we must set AVID's word size environment variable to equal this value. We also set AVID's endianness environment variable based on the architecture's endianness. On Unix systems, these environment variables can be set with the export command, as shown in Listing 5.3.

```
export AVID_ARCH_SIZE="64"
export AVID_ENDIANNESS="LE"
```

Listing 5.3: Setting AVID environment variables.

Finally, we can invoke the API of the generated parser. Suppose we have a byte slice that we want to interpret as a `Syscall` struct. Listing 5.4 shows how the `parse_and_create` method—automatically generated by AVID—is invoked with the byte slice.

```
let input = [...];
let syscall = Syscall::parse_and_create(input);
```

Listing 5.4: Invocation of AVID's exported `parse_and_create` method on a slice of input bytes.

Here, `parse_and_create` first validates the input and then instantiates and returns a `Syscall` struct containing this input. Using our system, we can be sure that

the returned struct represents a correct cast of the input bytes.

5.5 Integration with Tock

To fulfill our goal of securing input validation in Tock, we integrated the AVID toolchain into the Tock kernel, invoking the parsing mechanism in various subroutines that handle input. AVID integration was as simple as adding it as a Cargo dependency of the Tock kernel as well as configuring AVID's necessary environment variables in Tock's Makefile. Applying the toolchain to `parse_and_validate_tbfheader`, where raw pointers into flash memory are read from into application header structs, was a simple and effective way to add a layer of verification to the long and complex parsing method.

AVID also proved useful for a couple of other Tock functionalities. Running Tock on a board from Nordic Semiconductor's nRF52 series, we used AVID in Tock's nRF52-specific library to verify a struct of system call arguments—`usize` primitives passed from user space via the board's memory-mapped registers—for Tock's Command system call. Using AVID to ensure that `usize`'s are read as the correct size is especially useful in this case, as reading out-of-bounds from memory-mapped registers could result in a dangerous spatial error within sensitive kernel memory.

Lastly, we integrated AVID with Tock's Bluetooth Low Energy (BLE) advertising tools within the userland library libtock-rs. While AVID is not designed to parse a full Bluetooth packet, which takes the form of a length field and a variable-length payload, we used it to securely parse the packet header. This validated header can then be safely used to read the rest of the payload. This same Bluetooth header parsing can be added to Tock's Bluetooth-related capsules, but for our purposes, it was easier to test the integration by sending packets from userland.

Many projects targeted at embedded systems face unique implementation challenges related to platform and architecture constraints, and AVID is no different. One Rust-related issue frequently encountered by developers of embedded systems is lack of support for Rust's standard library. Rust's standard library contains commonly

used types and functionality that make life easier for Rust developers. However, since some of its tools contain architecture-specific operations such as file I/O, it is not supported by all targets, including Tock's. Therefore, integrating AVID with Tock involved modifying the AVID codebase and all dependencies to use Rust's `no_std` configuration.

A more unique issue to integration with Tock specifically is the absence of a heap within the Tock kernel. Not being able to dynamically allocate memory during parsing provided interesting challenges for the packrat parsing algorithm, which uses a stack to keep track of current rules. While the original Pest implementation used a Rust `Vec` to record this information, since `Vec`'s are dynamically allocated, this was not possible for our integration. We instead used a fixed-size array of size 50 for the stack. This fixed-size array constrains the number of nested rules to 50, which should be more than sufficient for simple kernel data structures.

Chapter 6

Evaluation: Security

In embedded systems with a variety of potentially untrusted peripheral devices, correctly validating external input is critical. Tock aims to bring security to embedded systems by adopting an isolated kernel design as well as leveraging Rust's security features. However, even Tock is not immune to bugs and design errors. In June 2018, the publication of CVE-2018-1000660 revealed a bug within `parse_and_validate_tbfheader` such that a compromised Tock capsule would be able to access arbitrary memory [23]. While this vulnerability is no longer an issue after the deprecation of `TbfHeader` v1, it reveals the very real possibility that there are further input handling errors in Tock.

The reason for this vulnerability is not a parsing or casting error but rather a design error combined with side-effects of `unsafe` Rust. Unfortunately, AVID is not equipped to protect against this design error. The landscape of input handling-related exploits is so broad and diverse that it is difficult if not impossible for a single, comprehensive solution to protect against every improper input validation vulnerability. Threat models constantly change as technology evolves and is used in new ways.

Therefore, AVID is not intended to be a catch-all solution for arbitrary input validation errors. It is instead designed to handle a specific set of issues that Rust developers face when implementing input handling routines. Specifically, AVID is able to prevent spatial errors caused by incorrect pointer arithmetic; spatial errors

caused by reading out-of-bounds memory; and type confusion errors during struct instantiation, including those due to erroneous architectural details (i.e., endianness, word size). Such dangerous errors may seem simple to an experienced developer, but complex systems with a large amount of input handling code often contain such bugs.

6.1 Tock Vulnerability Example

We now consider a manufactured vulnerability within Tock that can be avoided by enabling AVID. In Tock, system call arguments are read from memory-mapped registers by incrementing a stack pointer. Since it is possible for system call arguments to be pointers, all arguments are of the Rust `usize` primitive type—`usize`'s, like pointers, have length equal to the word size of the given architecture.

However, a careless developer may not realize that the length of a `usize` depends on the platform. Instead, they may assume that the length is the same as on their personal machine, perhaps 64 bits. On the other hand, small, portable boards like those from the nRF52 series usually use only 32-bit words. Mistakenly reading 64-bit integers instead of 32-bit integers will cause a spatial error such that kernel memory following the memory-mapped registers will be interpreted as `usize` data. Therefore, depending on how many system call arguments are being accessed, the careless developer will introduce a dangerous bug that leaks kernel memory.

We introduced this bug to the Tock kernel to demonstrate how such an error could be abused. We created a simple user application that invoked a system call from Tock's Bluetooth advertising driver capsule. This system call, which sets the transmitting power of the driver, accepts a single argument: the desired transmitting power in milliWatts [24]. However, the erroneous kernel code will not pass to the capsule the user's desired power but rather the contents of kernel memory from just past the memory-mapped registers. Figure 6-1 shows the results of this error—the capsule attempts to set the transmitting power to the exposed contents of kernel memory. Although the driver is not able to support every possible provided transmitting power, in certain cases, an adversary could perform a side-channel attack and

read kernel memory by measuring the transmitting power.

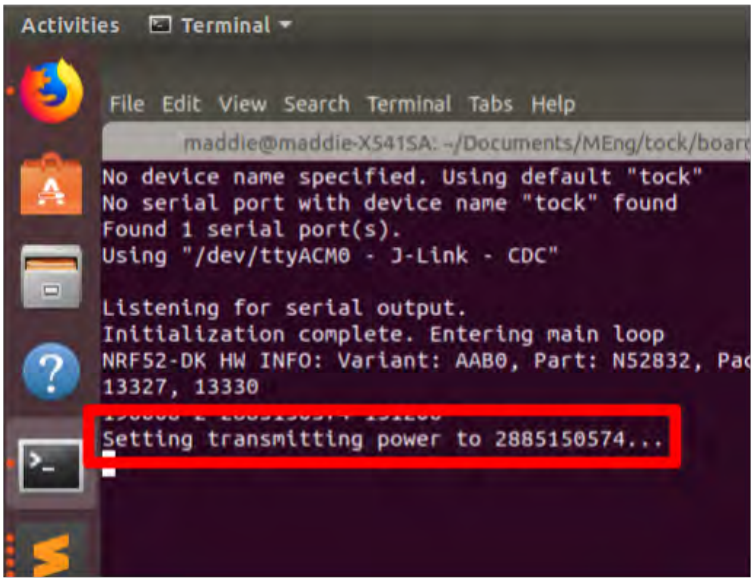

Figure 6-1: Demonstration of our manufactured Tock vulnerability.

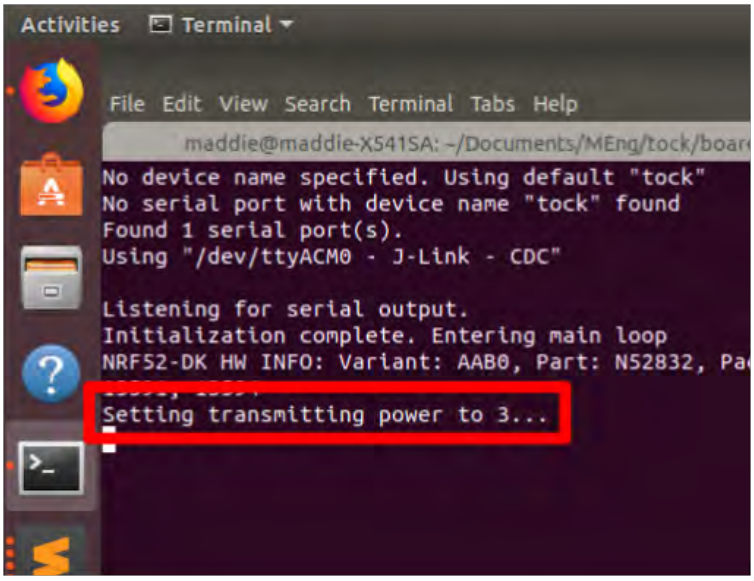

Figure 6-2: Demonstration of correct behavior after applying AVID.

We then applied AVID in an attempt to fix the system call struct bug. Replacing the incorrect parsing code with an invocation of `parse_and_create`, we see in Figure 6-2 that the vulnerability is resolved—the user's desired transmitting power, not contents of kernel memory, are passed to the capsule. With AVID in use,

the careless developer no longer needs to manually read system call arguments from memory. Furthermore, AVID keeps track of architectural details using the correctly set environment variables, so the developer has no reason to be mistaken about the architecture's word size in the first place.

6.2 Out-of-Scope Threats

We now pivot to consider specific issues AVID is not designed to solve and discuss our reasoning behind these decisions. Firstly, AVID supports neither packet types nor type-length-value (TLV) data and thus is not able to ensure correct length-based payload parsing. There are several reasons for this decision. PEGs, which are context-free, do not allow for length fields or other special fields that determine how the rest of the input is interpreted. Since we wish for AVID to benefit from the unambiguity guarantees of PEGs, the system is not able to support such fields.

Furthermore, allowing such fields would require us to change AVID's type definition interface. The fact that AVID reads type information from the Rust struct definition at compile time means that developers do not have to put in extra effort to specify struct grammars. Changing the interface to allow for special fields would complicate the system and increase developer effort to enable AVID. On the other hand, a developer that parses a packet header using AVID can confidently use the parsed length field to read the packet payload.

Another decision we made when designing AVID's scope of defense was to keep the system specific to Rust. Some parser generators, like Kaitai, are language-agnostic; they help solve the problem of type consistency across multiple platforms [25]. Since type information is lost when data moves past a system boundary, correctly reconstructing this type once it reaches its destination is crucial lest a type confusion error arise. Tools like Kaitai assist with this process by keeping track of type information in a central location that can be accessed by Kaitai toolchains on multiple platforms.

However, since AVID leverages Rust safety guarantees, we decided not to extend AVID to other languages without such guarantees (e.g., C). Furthermore, since

AVID is designed to be enabled within kernels, which are usually unilingual, we have no need to implement the toolchain for multiple languages. Lastly, extending AVID to multiple languages would require developers to specify type grammars in a language-agnostic DSL, burdening them with extra work when enabling AVID in their codebases.

Chapter 7

Evaluation: Performance

We now evaluate and discuss the performance overhead of AVID. We not only present microbenchmark results for both compile- and runtime but also consider AVID's performance implications for representative use cases of the toolchain within Tock. Evaluation code is included in the appendix.

7.1 Microbenchmarks

Microbenchmark experiments were run on an Intel Celeron N3060 CPU outside of Tock. Microbenchmark measurements were performed outside of Tock for increased timer precision. We applied AVID to a bare-bones Rust program containing only the necessary import statements and struct definition(s). The program, provided in Listing A.4, was compiled and run with Rust nightly compiler 1.31. We performed three sets of microbenchmark measurements, regarding program compile time with and without AVID under various struct configurations, input parsing time, and struct instantiation time.

Compile time was measured as the time to compile only the Rust file containing our simple program. Starting with a struct definition containing three fields of `u16`'s (Rust's unsigned 16-bit integer primitive), we compared the compile time of the program when AVID parser generation was enabled versus disabled. Figure 7-1 presents our results. Measuring elapsed time using Unix's `date` command, we saw that en-

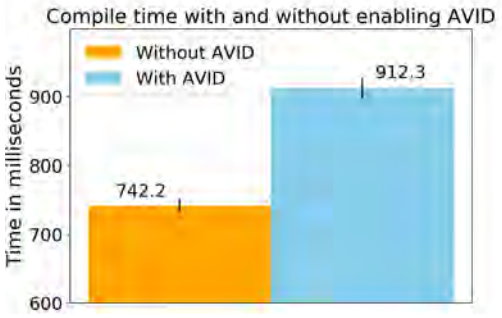

Figure 7-1: Compilation time with and without AVID parser generation.

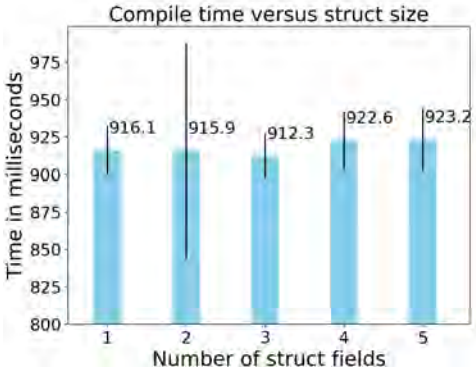

Figure 7-2: Compilation time versus number of struct fields.

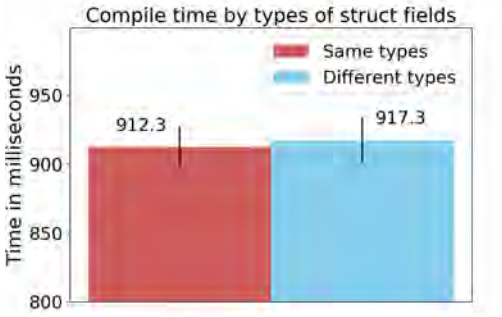

Figure 7-3: Compilation time by types of struct fields.

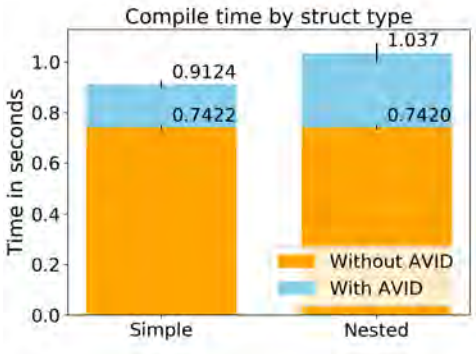

Figure 7-4: Compilation time for simple versus nested structs.

abling AVID parser generation for our three-field struct added 170.15 milliseconds to the original 742.16 ms, for a total of 912.31 ms—an approximate 23% overhead.

We next considered how struct size, or number of struct fields, affected compilation time. For this experiment, we varied the number (but not the type) of fields in our original struct. During parser generation, AVID outputs reader methods for each primitive type used, which are later invoked during struct instantiation. Since our structs contained only `u16`'s, we see from the results in Figure 7-2 that the amount of code generated by AVID did not significantly change with the number of struct fields containing `u16`'s.

A logical follow-up to this experiment is to test whether a struct with more diverse field types would take longer to compile than a struct containing fields of the same type. We modified our original three-field struct by changing one of the three field

types to an `u8` (unsigned 8-bit integer) and another field to an `i16` (signed 16-bit integer). Figure 7-3 compares the compilation time for this new struct definition compared to the original definition. Although a struct with more diverse types requires that AVID generate more code, there turns out to be no significant increase in compilation time.

Finally, we explored whether AVID's parser generation took significantly more time for nested structs. For this measurement, we modified the program to contain a second struct definition, one of whose fields was our original struct type. We measured compilation time for the simple and nested structs both with and without enabling AVID parser generation. Results are shown in Figure 7-4. We see that while without AVID, compilation time for the two types of structs is not significantly different, enabling AVID causes a significant increase in compilation for the nested struct. The reason for this difference is that the AVID must generate additional parsing and casting functionality for the outer struct to correctly handle the inner struct.

We now move on to consider how varying struct definitions affect the time for AVID to parse input. Specifically, using Rust's `time` library, we measured the time elapsed while AVID's exported parse method validated provided input bytes. The time for the parse function to complete is largely the time for Pest's packrat parser implementation to validate the input.

We first returned to considering the effect of number of struct fields, again all `u16`'s. Intuitively, a larger struct should take longer to parse, as more input must be checked. Figure 7-5 confirms this suspicion—larger structs take longer to parse. On average, extending a struct definition to contain an additional `u16` increased the time to parse by approximately 822.13 nanoseconds.

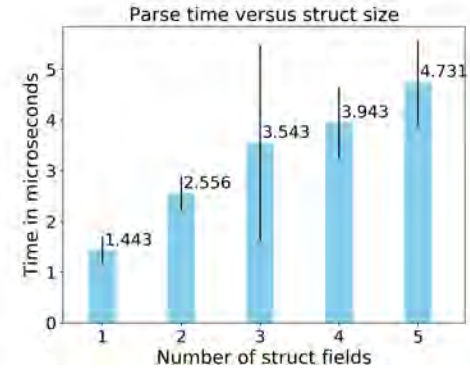

Figure 7-5: AVID parse time versus number of struct fields.

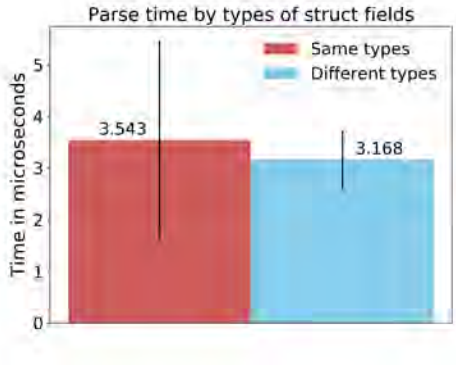

Figure 7-6: AVID parse time by types of struct fields.

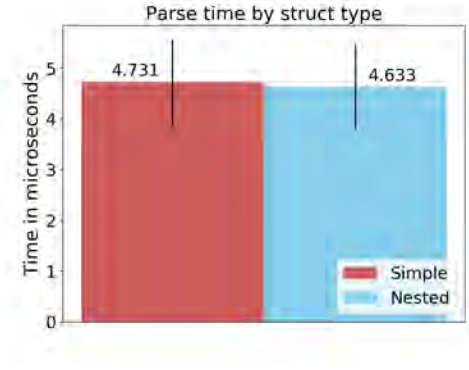

Figure 7-7: AVID parse time for nested versus five-field structs.

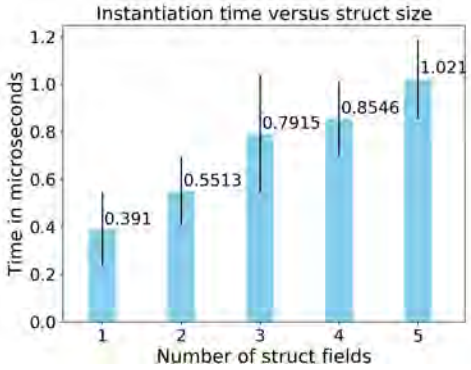

Figure 7-8: AVID instantiation time versus number of struct fields.

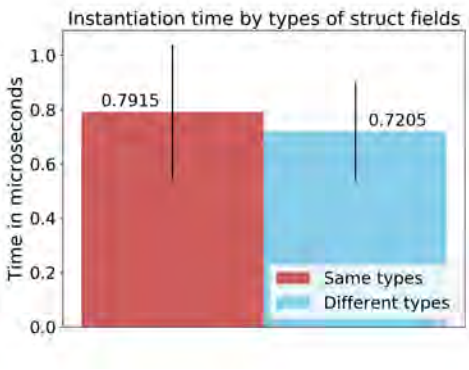

Figure 7-9: AVID instantiation time by types of struct fields.

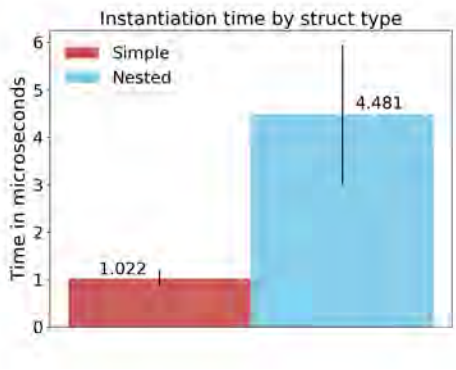

Figure 7-10: AVID instantiation time for nested versus five-field structs.

We repeated the above experiments regarding field type diversity and nested structs while measuring parse time. Changing the original struct of three `u16`'s to a struct containing a `u8`, a `u16`, and an `i16` did not significantly affect parse time, as shown in Figure 7-6. To measure the effect of nested structs on parse time, we compared a simple struct containing five `u16`'s to the original three-field struct nested within a parent struct with an additional `u16` field, for a total of five fields. We again found that there was no significant difference in parse time for these two struct definitions, as shown in Figure 7-7.

Lastly, we considered the time taken to instantiate structs using AVID's novel instantiation functionality. The instantiation routine does not include input validation and is intended for use only on previously validated input. We repeated the same measurements as completed above for the parse time experiments, considering the effects of struct size, field types, and struct nesting.

We again found that larger structs took longer to instantiate. Results are shown in Figure 7-8. Each additional `u16` increased the time to instantiate by an average of approximately 157.66 nanoseconds. This result should again be intuitive, as the amount of work to instantiate a struct increases linearly with the number of fields. Also similar to our parse measurement results, we see in Figure 7-9 that our struct with more diverse fields did not take a significantly different amount of time to instantiate compared to the original three-field struct.

On the other hand, unlike our parse time results, instantiating a nested struct took significantly longer than a simple struct with the same number of fields. Results are shown in Figure 7-10. For the nested struct, instantiation time increased from 1021.6 nanoseconds to 4481.0 nanoseconds—a 3458.4-nanosecond or approximately 338% increase. The reason for this is that the instantiation method for the outer struct must validate the bytes corresponding to the inner struct; therefore, the instantiation routine for the outer struct includes the parsing routine for the inner struct.

7.2 Tock Integration

After completing our AVID integration with Tock, we considered the performance implications of enabling the toolchain. Running Tock on Nordic Semiconductor's nRF52832 board, compiled with Rust nightly compiler 1.30, we chose certain regions of the code base that handle external input to implement representative applications of AVID. These applications include application header parsing, system call argument validation, and Bluetooth packet header construction.

Our first application is our original motivating example for integrating AVID into Tock: the parsing and validation of a `TbfHeader`. On startup, Tock loads application metadata from flash memory into `TbfHeader` structs. Our goal is to use AVID as a level of verification over Tock's long and complex `parse_and_validate_tbfheader` function. We enabled AVID's validation routing on two sub-structs of `TbfHeader`.

Enabling AVID parser generation and validation for these two structs, we measured the difference in application load time (for a single application) at startup. Results are shown in Figure 7-11. This procedure was repeated 100 times by running a custom script that repeatedly reloaded the application onto the board. Time elapsed was measured in number of clock ticks, directly using the on-board clock. This clock, whose frequency is 32.768 kHz [26], gives us a measurement granularity of approximately 30.5 microseconds.

We can see that enabling AVID validation within `parse_and_validate_tbfheader` increased the number of clock ticks spent on loading the application from 3 to 11—an overhead of approximately 244 microseconds or 267%. We ran the same experiment two more times, each time disabling parsing for one of the two structs, and found that the time to parse and instantiate each struct was approximately equal: 4 clock ticks.

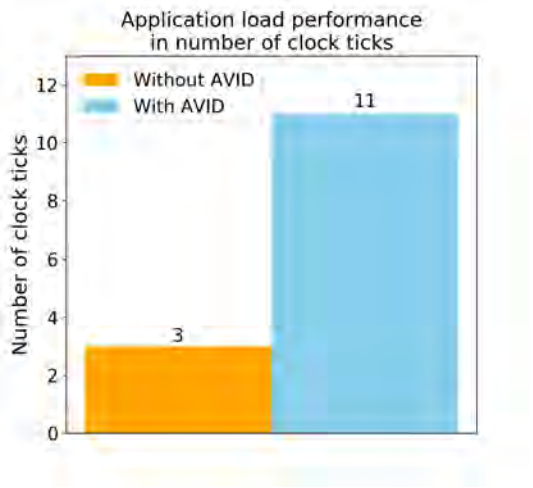

Figure 7-11: Time to load applications in Tock with and without AVID enabled.

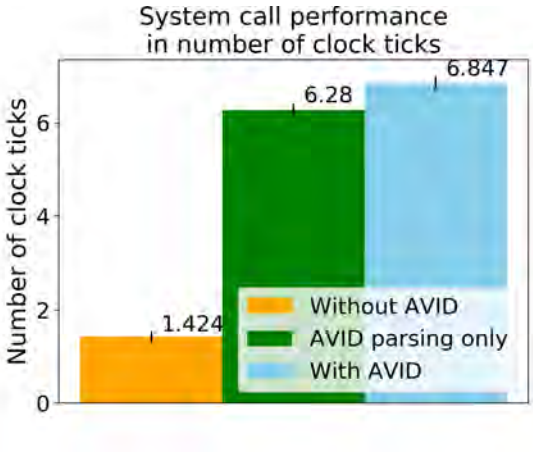

Figure 7-12: Time to perform a system call in Tock with and without AVID enabled.

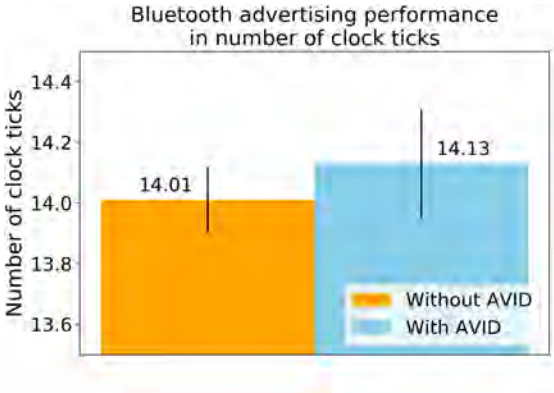

Figure 7-13: Time to send Bluetooth advertising packets in Tock with and without AVID enabled.

We next measured the overhead for applying AVID to system call argument validation. System call arguments are passed as `usize` fields from userland and then read by the kernel from memory-mapped registers. Using AVID to ensure that the `usize`'s are correctly read is critical here, as accidentally reading the nRF52's 32-bit `usize` primitives as 64-bit would cause a spatial error that exposes kernel memory.

For this example, we created a user application using libtock-rs that repeatedly measured the time for batches of ten system calls to complete. Time elapsed was again measured using the on-board clock, this time via the system call interface exposed to userland. In order to further investigate the significant overhead imposed by enabling AVID, we performed three different measurements: completion time with AVID disabled, completion time with only AVID parsing enabled, and completion time with both AVID parsing and instantiation enabled.

Figure 7-12 shows us that there was again a significant performance overhead for enabling AVID validation. We saw that the bulk of AVID's overhead is imposed by its parsing functionality. After enabling parsing, the average number of clock cycles spent per system call increased from 1.424 to 6.280—an overhead of approximately 148.2 microseconds or 341%. Enabling struct instantiation on top of parsing only incurred a further overhead of approximately 0.567 clock cycles per system call—an additional overhead of approximately 17.3 microseconds or 39.8%.

Our final example Tock integration is with libtock-rs functionality that constructs Bluetooth advertisement packets. There are many places we could have enabled Bluetooth header parsing, including Bluetooth-related Tock capsules as well as architecture-specific networking routines. We chose to integrate AVID with libtock-rs' Bluetooth tools for ease of packet creation and performance measurement. Time elapsed was again measured with the on-board clock via system call from userland.

Figure 7-13 shows us that the performance overhead for this application was not as significant as for the previous examples. Time to send an advertisement packet only increases from 14.01 to 14.13 clock ticks when AVID is enabled—an increase 0.85%. One possible reason for the overhead being much smaller in this example is that the struct being parsed is much smaller, containing only three bytes.

7.3 Discussion

We now further discuss the significant performance overhead imposed by AVID's parsing functionality. To do so requires first examining certain details of the parser implementation that AVID leverages. As AVID invokes its `parse` method on the provided input data, it saves the current parser state as a `ParserState` struct. This struct contains certain metadata such as the parser's current position within the input as well as the current rule being examined.

Since rules can be nested, AVID saves a list of the current rules as a stack. In the original Pest implementation, this stack was represented by a `Vec`. However, since AVID is not able to support dynamically allocated data structures (including `Vec`'s), in AVID, this stack is instead represented by a fixed-size array, as described in section 5.5.

Sequences of rules are implemented as sequences of nested closures. Each closure accepts a `ParserState` and then passes to a method of `ParserState` a closure that accepts the same `ParserState`. This pattern results in the same instance of `ParserState`—including the contents of its stack field—being repeatedly pushed onto and then popped off of the current call stack.

This repeated passing of the `ParserState`'s stack results in significant slowdown of AVID's parsing functionality. Possible solutions to this issue include decreasing the size of the fixed-size array that represents the `ParserState` stack; rethinking parser internals to not pass around instances of `ParserState` as function arguments; and statically allocating the stack so that only one instance of it exists.

Chapter 8

Related Work

In this chapter we review previous approaches to input validation for memory safety and compare them to AVID. There are many approaches to secure input handling in complex systems. Methods orthogonal to AVID include tagged architectures, information flow control, and software compartmentalization [27] [28] [29]. However, in this section we choose to focus on parser generators and other input validation techniques with functionality more similar to AVID. Qualities being compared include use cases, scopes of defense, and usability.

8.1 Input Validation Techniques

We first consider previous work on parsing solutions for data types and binary formats. Qualities being compared are presented in Table 8.1. Motivated by input injection attacks, Hermerschmidt et al. [30] describe a system that is able to safely and correctly encode input for use in a document described by a context-free grammar. While the system is evidently successful in preventing XSS attacks, its scope is too limited for our intended use case. AVID goes beyond this system by parsing and instantiating data structures at runtime as well as limiting the grammar to ambiguity-free PEGs.

Hammer is a parser generator for binary formats. Hammer generates packrat parsers, the same linear-time, memoized parsers used by AVID to parse PEGs [19]. However, Hammer's approach is different from AVID in that Hammer uses parser

	Features and attributes			
	Ambiguity-free	Packrat parser	Instantiation	Supports Rust
AVID	✓	✓	✓	✓
Hermerschmidt et al.	✗	✗	✗	✗
Nail	✓	✓	✓	✗
McHammerCode	✓	✗	✓	✗

Table 8.1: Comparison of AVID and related input validation solutions.

combinators to build the final parser out of multiple sub-parsers. Unlike AVID and Kaitai, Hammer is only able to parse input and cannot instantiate a struct based on the input. Additionally, Hammer does not support Rust code.

Nail is a binary format parser generator that extends Hammer with an API that is more useful to developers [31]. Specifically, Nail advertises that it is able to output structs from parsed input and that it supports length field matching. Although grammar operators in Nail behave similarly to PEG rules (e.g., ordered choice, greedy Kleene star), Nail does not use a packrat parser but rather a simpler and less efficient top-down parser. Also unlike PEGs, Nail's grammar is not context-free—developers can include length fields that influence how input is parsed. This behavior is useful in certain contexts (e.g., network packets), but dangerous in that attackers have control over this field (e.g., the Heartbleed attack). Furthermore, the fact that Nail is written in C leaves something to be desired for our intended use case.

McHammerCode is an encoder that is also based on Hammer [32]. Like [30], McHammerCode is motivated by and intends to prevent input injection attacks. McHammerCode goes beyond [30] in that like AVID, it is able to create language primitives (e.g., ints, floats) from parsed input and return them to the user. Unfortunately, McHammerCode's grammar is dangerously powerful. Not only does it allow length fields, but also offset rules that cause the program to access memory at calculated but arbitrary offsets based on user input.

One common use case for manually parsing data structures at runtime is in the context of networking. Standardized packet types cause developers to include hard-coded input validation and handling in their networking applications. Correct handling of this third-party data is critical since it is untrusted and possibly malicious, and there have been many attempts to secure networking code with parser generators [33] [34] [35] [36] [37] [38] [39]. However, such parsers are designed for different scenarios from AVID and therefore optimized for different constraints. For example, while packet parsers must be able to handle variable-length packet types, AVID distrusts types with attacker-controlled length fields and does not support them.

8.2 Parser Generators

AVID's parsing mechanism leverages a parser generator. When developing the system, instead of rolling our own from scratch, we decided to integrate an existing implementation. While we ended up choosing to leverage Pest, there were several other contenders that we initially considered: Oak [40], Kaitai [25], and Nom [41].

Oak is a PEG parser generator written in Rust and intended to be used for parsing ASCII characters. Unlike Pest, Oak supports semantic actions: context-dependent rules that can affect how future input is parsed. While Pest generates parsers using procedural macros, Oak instead employs regular Rust macros to create parsers—in Oak's case, the grammar is simply the tokens passed into the macro invocation. Oak's interface is therefore undesirable for our use case; we prefer for Rust struct definitions to act as grammars. Furthermore, at the time this thesis was written, Oak did not support all valid PEGs, including certain recursive grammars.

Kaitai generates parsers for binary formats described by Kaitai Struct, its DSL. Because Kaitai grammars are defined in this DSL, commonly-used data-types can be defined just once and then parsed on multiple platforms. Kaitai Struct is much more expressive than PEGs and can express types such as TLV data, where the value of a length field specifies the size of the type. This expressiveness is useful, but it also makes Kaitai parsers less performant than PEG parsers. While Kaitai offers runtime

libraries for a variety of languages, Rust is currently not one of them.

Similar to Hammer, Nom uses parser combinators to build parsers for binary formats. Nom is written in and intended for use in Rust. Unlike Hammer, Nom is able to instantiate structs from parsed input. Furthermore, unlike both Hammer and AVID, Nom is able to parse context-dependent TLV types. One attractive feature of Nom is that the library is functional in a `no_std` context without modifications. Nom is also able to be integrated with C applications with an exported C API [42].

Chapter 9

Conclusion

AVID is an original input validation solution for Rust that generates parser implementations that safely parse and instantiate structs from untrusted input. Integrating AVID with existing systems requires minimal developer effort—to trigger parser generation, developers only need to add a simple source code annotation above each struct. Invoking the parsing and instantiation routines is as simple as passing a byte slice to AVID's generated parser API. Architectural details can be configured in a central location using environment variables so that the average developer need not concern themselves with such specifics.

AVID prevents certain types of critical type confusion and spatial errors. Together, AVID's parsing and instantiation routines ensure that input is validated and then correctly cast to the desired struct type. In our exploit evaluation, we showed how AVID can be used to successfully prevent a type confusion error related to incorrect `usize` length. In our performance evaluation, we found that parsing and instantiation times intuitively scale with the size of the struct, or number of fields.

AVID's implementation leverages Pest as well as Rust's procedural macro system. Pest, a PEG parser generator, provides AVID's generated parsers with unambiguity guarantees. To uphold these guarantees, AVID restricts what types of structs are able to be parsed—packet types are not supported, as the influence that the length field has over struct parsing requires that grammars not be context-free. Instead, AVID is best suited for simple kernel structs containing Rust primitives, and its unambiguity

and safety guarantees make it a helpful tool for this use case.

9.1 Future Work

One obstacle we faced when enabling AVID's parser generation for Tock's `TbfHeader` struct was that one of the sub-structs contained a string field. While C strings are relatively simple to parse because of null termination, Rust strings are not null-terminated—parsing a Rust string requires extra information beside the provided input. In the future, it would be beneficial to implement parsing of C-formatted strings. However, this would need to be executed in such a way such that there is no possibility for confusion or ambiguity between C- and Rust-formatted strings.

We would also like to add support for other types of Rust types: enums and tuple structs. Parsing and instantiating these data types is conceptually the same as doing so for structs. The only hurdle for modifying AVID to support them is to add functionality to the procedural macro that can distinguish struct, enum, and tuple struct definitions; gather field information using their respective `syn` API's; and then generate parser implementations accordingly.

Runtime performance was not an immediate priority while implementing AVID, and the significant overhead incurred by enabling AVID in Tock indicates that there is certainly room for improvement. While parsing and instantiation take only a few microseconds on modern laptop processors, this extra overhead can become substantial for smaller systems like the nRF52 series. Especially for high-throughput applications of AVID such as system calls and packet header parsing, it is important to keep in mind the resource constraints of embedded systems and optimize accordingly.

Appendix A

Code References

A.1 Security Evaluation Code

In this section, we present the code used to perform the security evaluation discussed in 6.1. The original Tock implementation is also provided in Listing A.1 for reference.

Note that in Listing A.3, system call arguments are first parsed using AVID into a separate struct; the fields of this struct are then copied into the `COMMAND` enum instance. The reason for this is that AVID currently cannot accept enum definitions as grammars. Resolving this issue is an area for future work, as mentioned in section 9.1.

```
/// Get the syscall that the process called.
unsafe fn get_syscall(
    &self,
    stack_pointer: *const usize
) -> Option<kernel::syscall::Syscall> {
    // Get the four values that are passed with the syscall.
    let r0 = read_volatile(stack_pointer.offset(0));
    let r1 = read_volatile(stack_pointer.offset(1));
    let r2 = read_volatile(stack_pointer.offset(2));
    let r3 = read_volatile(stack_pointer.offset(3));
    // ...
    let svc_num = [...];
    match svc_num {
        // ...
        2 => Some(kernel::syscall::Syscall::COMMAND {
            driver_number: r0,
            subdriver_number: r1,
            arg0: r2,
            arg1: r3,
        }),
        // ...
    }
    // ...
}
```

Listing A.1: Original Tock system call struct parsing routine.

```
/// Get the syscall that the process called.
unsafe fn get_syscall(
    &self,
    stack_pointer: *const usize
) -> Option<kernel::syscall::Syscall> {
    // Get the four values that are passed with the syscall.
    let r0 = read_volatile(stack_pointer);
    let r1 = read_volatile((stack_pointer as *const u64).offset(1))
        as usize;
    let r2 = read_volatile((stack_pointer as *const u64).offset(2))
        as usize;
    let r3 = read_volatile((stack_pointer as *const u64).offset(3))
        as usize;
    // ...
    let svc_num = [...];
    match svc_num {
        // ...
        2 => Some(kernel::syscall::Syscall::COMMAND {
            driver_number: r0,
            subdriver_number: r1,
            arg0: r2,
            arg1: r3,
        }),
        // ...
    }
    // ...
}
```

Listing A.2: Tock system call struct parsing routine with manufactured vulnerability.

```
#[allow(dead_code)]
#[derive(Parser)]
struct Command {
    driver_number: usize,
    subdriver_number: usize,
    arg0: usize,
    arg1: usize,
}

/// Get the syscall that the process called.
unsafe fn get_syscall(
    &self,
    stack_pointer: *const usize
) -> Option<kernel::syscall::Syscall> {
    let input =
        slice::from_raw_parts(stack_pointer as *const u8, 4 * 64);
    let command = Command::parse_and_create(input);
    // ...
    let svc_num = [...];
    match svc_num {
        // ...
        2 => Some(kernel::syscall::Syscall::COMMAND {
            driver_number:      command.driver_number,
            subdriver_number:   command.subdriver_number,
            arg0:               command.arg0,
            arg1:               command.arg1,
        }),
        // ...
    }
    // ...
}
```

Listing A.3: Tock system call struct parsing routine vulnerability solved with AVID.

A.2 Performance Evaluation Code

This section contains code run and code snippets used to perform performance evaluations.

A.2.1 Microbenchmarks

```
extern crate core;
extern crate avid;
#[macro_use]
extern crate avid_derive;

use avid::Parser;
use std::time::Instant;

#[derive(Parser)]
struct MyStruct {
    first_field: u16,
    second_field: u16,
    third_field: u16,
}

fn main() {
    for _ in 0..200 {
        let start = Instant::now();

        let input = [...];
        let t = MyStruct::parse_and_create(&input);

        let end = start.elapsed();

        println!("{}", end.subsec_nanos());
    }
}
```

Listing A.4: Microbenchmark experiment environment.

```
#[derive(Parser)]
struct MyStruct {
    first_field: u16,
    second_field: u16,
    third_field: u16,
}
```

Listing A.5: Three-field microbenchmark struct.

```
#[derive(Parser)]
struct MyStruct {
    first_field: u8,
    second_field: u16,
    third_field: i16,
}
```

Listing A.6: Microbenchmark struct with differing field types.

```
#[derive(Parser)]
struct MyStruct {
    first_field: u16,
    second_field: u16,
    third_field: u16,
}

#[derive(Parser)]
struct ParentStruct {
    first_field: MyStruct,
    second_field: u16,
}
```

Listing A.7: Nested microbenchmark struct.

A.2.2 Tock Integration

```
1  /// TBF fields that must be present in all v2 headers.
2  #[repr(C)]
3  #[derive(Clone, Copy, Debug, Parser)]
4  crate struct TbfHeaderV2Base {
5      version: u16,
6      header_size: u16,
7      total_size: u32,
8      flags: u32,
9      checksum: u32,
10 }
11
12 /// The v2 main section for apps.
13 ///
14 /// All apps must have a main section. Without it, the header is
       considered as
15 /// only padding.
16 #[repr(C)]
17 #[derive(Clone, Copy, Debug, Parser)]
18 crate struct TbfHeaderV2Main {
19     init_fn_offset: u32,
20     protected_size: u32,
21     minimum_ram_size: u32,
22 }
```

Listing A.8: `TbfHeader` structs with AVID enabled.

Listing A.9 displays the Tock user application used to measure system call performance with and without AVID integration. AVID system call validation was as presented in Listing A.3. System call performance was measured in batches of ten to improve timing granularity.

```
#![no_std]
#![feature(alloc)]

extern crate alloc;
extern crate tock;

use alloc::string::String;
use tock::console::Console;
use tock::led;
use tock::timer;

fn main() {
    // Timer initialization
    let mut with_callback = timer::with_callback(|_, _| {});
    let timer = with_callback.init().unwrap();

    let mut console = Console::new();
    let led = led::get(0).unwrap();

    for _ in 0..100 {
        let start_time = timer.get_current_clock().num_ticks();

        for _ in 0..10 {
            led.on();
        }

        let end_time = timer.get_current_clock().num_ticks();

        let elapsed_time = (end_time - start_time) as u32;
        console.write(String::from("Total num ticks: "));
        console.write(tock::fmt::u32_as_decimal(elapsed_time));
        console.write(String::from("\n"));
    }
}
```

Listing A.9: User application for measuring system call performance.

```
#[derive(Parser)]
pub struct Flag {
    offset: u8,
    gap: u8,
    flag: u8
}
```

Listing A.10: Bluetooth advertisement packet flag struct.

```
#![no_std]
#![feature(alloc)]

extern crate alloc;
extern crate corepack;
extern crate tock;

use alloc::string::String;
use tock::console::Console;
use tock::ble_composer;
use tock::simple_ble;
use tock::timer;

fn main() {
    let mut console = Console::new();

    // Timer initialization
    let mut with_callback = timer::with_callback(|_, _| {});
    let timer = with_callback.init().unwrap();

    for _ in 0..100 {
        let start_time = timer.get_current_clock().num_ticks();

        for _ in 0..100 {
            let name = String::from("Tock!");
            let uuid: [u8; 2] = [0x00, 0x18];
            let str_payload = [...];
```

```
28              let payload = corepack::to_bytes(str_payload).unwrap();
29
30              let mut buffer = simple_ble::BleAdvertisingDriver::
                    create_advertising_buffer();
31              let mut gap_payload = ble_composer::BlePayload::new();
32
33              gap_payload.add_flag(ble_composer::flags::
                    LE_GENERAL_DISCOVERABLE);
34              gap_payload.add(ble_composer::gap_types::UUID, &uuid);
35              gap_payload.add(
36                  ble_composer::gap_types::COMPLETE_LOCAL_NAME,
37                  name.as_bytes(),
38              );
39              gap_payload.add_service_payload([91, 79], &payload);
40
41              let handle =
42                  simple_ble::BleAdvertisingDriver::initialize(
43                      100,
44                      &gap_payload,
45                      &mut buffer
46                  ).unwrap();
47          }
48
49          let end_time = timer.get_current_clock().num_ticks();
50
51          let elapsed_time = (end_time - start_time) as u32;
52          console.write(String::from("total num ticks: "));
53          console.write(tock::fmt::u32_as_decimal(elapsed_time));
54          console.write(String::from("\n"));
55      }
56
57      loop {}
58  }
```

Listing A.11: User application for measuring Bluetooth advertisement performance.

Bibliography

[1] "CWE-20: Improper input validation." https://cwe.mitre.org/data/definitions/20.html, Jan. 2019. Accessed Jan. 22, 2019.

[2] "CVE-2018-5007." https://www.cvedetails.com/cve/CVE-2018-5007, Sept. 2018. Accessed Jan. 22, 2019.

[3] "Adobe flash player type confusion error lets remote users execute arbitrary code and out-of-bounds memory read error lets remote users obtain potentially sensitive information." https://securitytracker.com/id/1041248, Jul. 2018. Accessed Jan. 22, 2019.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 48–62, IEEE Computer Society, 2013.

[5] "CWE-843: Access of resource using incompatible type ('type confusion')." https://cwe.mitre.org/data/definitions/843.html, Jan. 2019. Accessed Jan. 22, 2019.

[6] "Primitive type pointer." https://doc.rust-lang.org/std/primitive.pointer.html#method.sub. Accessed Jan. 22, 2019.

[7] N. Chomsky, "Three models for the description of language," *IRE Trans. Information Theory*, vol. 2, no. 3, pp. 113–124, 1956.

[8] J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference," in *IFIP Congress*, pp. 125–131, 1959.

[9] "Extensible markup language (XML) 1.1." https://www.w3.org/TR/xml11/, Sept. 2006. Accessed Jan. 22, 2019.

[10] "The go programming language specification." https://golang.org/ref/spec, May 2018. Accessed Jan. 22, 2019.

[11] "Gnu bison." https://www.gnu.org/software/bison/, 2014. Accessed Jan. 22, 2019.

[12] M. Sipser, *Introduction to the Theory of Computation*. Boston, MA: Course Technology, third ed., 2012.

[13] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation.," in *POPL* (N. D. Jones and X. Leroy, eds.), pp. 111–122, ACM, 2004.

[14] "Tock overview." `https://github.com/tock/tock/blob/master/doc/Overview.md`, Dec. 2018. Accessed Jan. 22, 2019.

[15] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, pp. 761–763, Aug. 1984.

[16] "Console." `https://github.com/tock/tock/blob/37a9ac4c15a7b2c1065645cb28bd9a4da5732315/doc/syscalls/00001_console.md`, May 2018. Accessed Jan. 22, 2019.

[17] "CWE-200: Information exposure." `https://cwe.mitre.org/data/definitions/200.html`, Jan. 2019. Accessed Jan. 22, 2019.

[18] "Pest. the elegant parser." `https://github.com/pest-parser/pest`, Jan. 2019. Accessed Jan. 22, 2019.

[19] B. Ford, "Packrat parsing: a practical linear-time algorithm with backtracking," Master's thesis, Massachusetts Institute of Technology, 2002.

[20] "Parser for rust source code." `https://github.com/dtolnay/syn`, Jan. 2019. Accessed Jan. 22, 2019.

[21] "Rust quasi-quoting." `https://github.com/dtolnay/quote`, Jan. 2019. Accessed Jan. 22, 2019.

[22] "Crate byteorder." `https://docs.rs/byteorder/1.3.1/byteorder/`, Jan. 2019. Accessed Jan. 22, 2019.

[23] "CVE-2018-5007." `https://nvd.nist.gov/vuln/detail/CVE-2018-1000660`, Sept. 2018. Accessed Jan. 22, 2019.

[24] "Bluetooth low energy advertising driver." `https://github.com/tock/tock/blob/master/capsules/src/ble_advertising_driver.rs`, Dec. 2018. Accessed Jan. 22, 2019.

[25] "Kaitai struct." `https://kaitai.io/`, 2018. Accessed Jan. 22, 2019.

[26] Nordic Semiconductor, *nRF52832 - Product Specification v1.0*, Feb. 2016.

[27] H. Shrobe, T. Knight, and A. de Hon, "Tiara: trust management, intrusion tolerance, accountability, and reconstitution architecture," tech. rep., Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Lab, Technical Report MIT-CSAIL-TR-2007-028, May 2007.

[28] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, (Washington, DC, USA), pp. 20–37, IEEE Computer Society, 2015.

[29] S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B. C. Pierce, H. Reubenstein, J. M. Smith, G. T. Sullivan, A. Thomas, J. Tov, C. M. White, and D. Wittenberg, "Safe: A clean-slate architecture for secure systems," in *Technologies for Homeland Security (HST) 2013 IEEE International Conference on*, (Waltham, MA, USA), pp. 570–576, 2013.

[30] L. Hermerschmidt, S. Kugelmann, and B. Rumpe, "Towards more security in data exchange: Defining unparsers with context-sensitive encoders for context-free grammars," in *2015 IEEE CS Security and Privacy Workshops*, pp. 134–141, 2015.

[31] J. Bangert and N. Zeldovich, "Nail: A practical tool for parsing and generating data formats," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 615–628, USENIX Association, 2014.

[32] T. Bieschke, L. Hermerschmidt, B. Rumpe, and P. Stanchev, "Eliminating input-based attacks by deriving automated encoders and decoders from context-free grammars," in *2017 IEEE Symposium on Security and Privacy Workshops*, pp. 93–101, 2017.

[33] R. Sommer, J. Amann, and S. Hall, "Spicy: A unified deep packet inspection framework for safely dissecting all your data," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, (New York, NY, USA), pp. 558–569, ACM, 2016.

[34] A. ElShakankiry, "Context sensitive and secure parser generation for deep packet inspection of binary protocols," Master's thesis, Queen's University, Kingston, Ontario, canada, 2017.

[35] A. Alim, R. G. Clegg, L. Mai, L. Rupprecht, E. Seckler, P. Costa, P. Pietzuch, A. L. Wolf, N. Sultana, J. Crowcroft, A. Madhavapeddy, A. W. Moore, , R. Mortier, M. Koleni, L. Oviedo, D. McAuley, and M. Migliavacca, "Flick: Developing and running application-specific network services," in *2016 USENIX Annual Technical Conference*, 2016.

[36] S. Mondet, I. Alberdi, and T. Plagemann, "Generating optimised and formally checked packet parsing code," in *Future Challenges in Security and Privacy for Academia and Industry - 26th IFIP TC 11 International Information Security Conference, SEC 2011, Lucerne, Switzerland, June 7-9, 2011. Proceedings*, pp. 173–184, 2011.

[37] O. Levillain, "Parsifal: A pragmatic solution to the binary parsing problems," in *Proceedings of the 2014 IEEE Security and Privacy Workshops*, SPW '14, (Washington, DC, USA), pp. 191–197, IEEE Computer Society, 2014.

[38] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo, "Generic application-level protocol analyzer and its language," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*, 2007.

[39] S. Suriyakarn, C. Pit-Claudel, B. Delaware, and A. Chlipala, "Narcissus: Deriving correct-by-construction decoders and encoders from binary formats," *CoRR*, vol. abs/1803.04870, 2018.

[40] "Oak." `https://github.com/ptal/oak`, Jan. 2019. Accessed Jan. 22, 2019.

[41] "Nom, eating data byte by byte." `https://github.com/Geal/nom`, Oct. 2018. Accessed Jan. 22, 2019.

[42] P. Chifflier and G. Couprie, "Writing parsers like it is 2017," in *2017 IEEE Security and Privacy Workshops, SP Workshops 2017, San Jose, CA, USA, May 25, 2017*, pp. 80–92, 2017.