Bouncer: Static Program Analysis in Hardware

Joseph McMahan
UC Santa Barbara
jemcmahan13@gmail.com

Michael Christensen
UC Santa Barbara
mchristensen@cs.ucsb.edu

Kyle Dewey
CSU Northridge
kyle.dewey@csun.edu

Ben Hardekopf
UC Santa Barbara
benh@cs.ucsb.edu

Timothy Sherwood
UC Santa Barbara
sherwood@cs.ucsb.edu

ABSTRACT

When discussing safety and security for embedded systems, we typically divide the world into software checks (which are either static or dynamic) or hardware checks (which are dynamic). As others have pointed out, hardware checks offer more than just efficiency. They are intrinsic to the device's functionality and thus are live from power-up; they require little to no dependency on other software functioning correctly, and due to the fact they are wired directly into the operation of the system, are difficult or impossible to bypass. We explore an experimental new embedded system that uses special-purpose hardware for static analysis that prevents all program binaries with memory errors, invalid control flow, and several other undesirable properties from ever being loaded onto the device. Static analysis often requires whole-binary-level, rather than instruction-level, examination. We show that a carefully constructed hardware state machine, using available scratch-pad memory, is capable of efficiently checking functional binaries in a streaming and verifiably non-bypassable way directly in hardware as they are loaded into the embedded program store. The resulting system is surprisingly small (taking no more than .0079 mm^2), efficient (capable of checking binaries at an average throughput of around 60 cycles per instruction), and yet guarantees execution free from many of the fragile behaviors that result in security and safety concerns. We believe this is the first time any static analysis has been implemented at the hardware level and opens the door to more complex hardware-checked properties.

ACM Reference Format:
Joseph McMahan, Michael Christensen, Kyle Dewey, Ben Hardekopf, and Timothy Sherwood. 2019. Bouncer: Static Program Analysis in Hardware. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307650.3322256

1 INTRODUCTION

The demand for more connectivity and richer interactions in everyday objects means that everything from light bulbs to thermostats now contains general-purpose microprocessors for carrying out fairly straightforward and low-performance tasks. Left unanalyzed, these systems and their associated software stacks can be expected to hold a seemingly endless collection of opportunities for attack. Static analysis provides powerful tools to those wishing to understand or limit the set of behaviors some software might exhibit. By facilitating sound reasoning over the set of all possible executions, this type of analysis can identify important classes of behavior and prevent them from ever happening. If embedded system developers simply *never* released software that failed, such that those well-analyzed applications were the *only* things to ever execute on platforms under our control, many of the bugs and vulnerabilities that plague our life would be eliminated. Unfortunately, realizing this in practice has proven incredibly hard due to pressure to market, pressure to reduce cost, and the delayed and stochastic cost associated with vulnerabilities and bugs.

While larger software companies might be more trusted to rigorously verify their software releases, the embedded systems market has a long and heavy tail of providers with a much wider distribution of expertise and resources at their disposal. When we bring an embedded device into our home or business, how can we have confidence that the software running there (which depends on chains of control well outside our ability to observe) is "above the bar" for us? Seemingly innocuous issues, for example passing a string instead of an integer, can open the door for an attacker to gain root privileges and serve as a base for other attacks (exactly this happened already in a class of WiFi routers [12]). Similar attacks targeting embedded devices and firmware updates have succeeded on everything from printers [11] to thermostats [18].

The basic research question we ask in this paper is: is it possible to make forms of static analysis an *intrinsic* part of executing on a microprocessor? In other words, we examine a machine that will *guarantee at the hardware level* that any and all code executing on it is bound to the constraints imposed by a given static program analysis. This moves the decision to do a proper analysis away from those that push software updates (who may be making decisions about updates many years removed from the original purchase) to the decision to purchase and deploy a particular hardware device itself.

Such a machine would reject any attempt to load it with code that fails to meet the specified "bar," independent of who wrote it, who signed it, how it was managed, or where the software came from. The trust one could put in aspects of execution on such a processor could be independent of measurement, attestation, or other active third-party evaluation. By doing the checks in hardware, we can make them intrinsic to the device's functionality: the checks will be fully live right from power-up; the checks will require no

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322256

dependency on other software on the system functioning correctly (zero TCB); and if properly designed, they will be directly wired into the operation of the system, making them provably impossible to bypass.

As this is the first approach to propose and evaluate fully-hardware implemented static analysis there are two big open questions: a) is it even possible to do a useful static analysis in hardware, and b) what would the costs of such an analysis be in terms of time or area? We answer these questions through the hardware development of a new module, the Binary Exclusion Unit (which we call "the bouncer" more informally), capable of scanning and rejecting program binaries right as they are streamed onto the device. Specifically, we make the following contributions:

- We introduce hardware static binary analysis and show that it can be implemented in a way that can never be circumvented through some clever manipulation of software (e.g. a compromised set of keys, a bug in the operating system, or a change in the boot ordering).
- We describe a method of static analysis co-design where the checking algorithm is modified to be more amenable to hardware implementation while maintaining correctness and efficiency.
- We demonstrate that the analysis, in conjunction with the functional ISA, ensures all executions are free of memory and type errors and have guaranteed control flow integrity.
- We evaluate the functioning of the system with a complete RTL implementation (synthesizable Verilog) of the checker and processor interoperating with gate-level simulation.
- Finally, we show that the resulting system is efficient both in terms of hardware resources required and performance, and describe how program transformations can make it even more so.

We elaborate on the motive of our work (Section 2), present our hardware static analysis in the form of a new hardware/software co-designed type system and prove its soundness (Section 3), outline the checking algorithm implementing the type system (Section 4), and design type annotations that can be easily encoded into the machine binary and provide a hardware implementation of the typechecker (Section 5). We prove the non-bypassability of the circuit in Section 6, something that would be extremely difficult to achieve for a software solution. Next, we provide hardware synthesis figures, evaluate update-time overhead, and show how to manage worst-case examples (Section 7). Finally, we discuss related work (Section 8) and conclude.

2 HARDWARE STATIC ANALYSIS

In building a static analysis hardware engine directly into an embedded micro-controller, one of the big advantages of customization is that at the hardware level we can see, *either physically through inspection or through analysis at the gate or RTL level*, exactly how information is flowing through a system to introduce safety or security mechanisms that are truly non-bypassable. No software can change the functioning of the system at that level. However, doing static analysis at the level of machine code is no easy task — even for software.

Fortunately, there are some great works to draw inspiration from. Previous work has used types to aid in assembly-level analysis; specifically TAL [22] and TALx86 [10] have created systems where source properties are preserved and represented in an idealized assembly language (the former) or directly on a subset of x86 (the latter). Working up the stack from assembly, other prior works attempt to prove properties and guarantee software safety at even higher levels of abstractions. We seek to take these software ideas and find a way to make them *intrinsic* properties of the physical hardware for embedded systems where needed.

In this work we draw upon the opportunity afforded by architectures that have already been designed with ease of analysis in mind. Specifically, we leverage the Zarf ISA, a purely functional, immutable, high-level ISA and hardware platform used for binary reasoning, which is suitable for execution of the most critical portions of a system [20]. At a high level, the Zarf ISA consists of three instructions: `Let` performs function application and object allocation, applying arguments to a function and creating an object that represents the result of the call. `Case` is used for both pattern-matching and control flow. One cases on a variable, then gives a series of patterns as branch heads; only the branch with the matching pattern is executed. Patterns can be constructors (datatypes) or integer values, depending on what was cased on. `Result` is the return instruction; it indicates what value is returned at the end of a function. Branches in case statements are non-reconvergent, so each must end in a result instruction.

A big advantage of this ISA for static analysis is that it has a compact and precise semantics. If we could could guarantee the physical machine would always execute only according to these semantics (e.g. always respecting call/return behavior, using the proper number of arguments from the stack, etc.) we would end up with a system that has some very desirable properties. In Section 7 we show that these include verifiable control flow integrity, type safety, memory safety, and others; e.g., ROP [4] is impossible, programs never encounter type errors, and buffer overruns can never happen.

Unfortunately, the semantics of any language govern the behavior of execution only for "well-formed" programs. When we are talking about machine code, as opposed to programming languages, things are a little trickier, because machines are expected to read instruction bits from memory and execute them faithfully as they arrive. As we describe in more detail below, checking membership in the language of well-formed Zarf programs is actually something that requires some sophistication and would be difficult to do at run-time. Even though there are just three instructions, Zarf binaries support casing, constructors, datatypes, functions, and other higher-level concepts as first-class citizens in the architecture. Our goal is to correctly implement these checks statically and show that the only binaries that can *ever* execute on this machine pass this static analysis.

2.1 The Analysis Implemented

While one could, in theory, capture every possible deviation from the Zarf semantics with a set of run-time checks in hardware, actually catching every possible thing that can go wrong quickly grows in complexity. An advantage of static checking over dynamic

Possible failure:	Meaning:
malformed instruction	Bit sequence does not correspond to a valid instruction.
fetch out-of-bounds arg	Accessing argument *N* when there are fewer than *N* arguments.
fetch out-of-bounds local	Accessing local *N* when there are fewer than *N* locals allocated.
fetch out-of-bounds field	Accessing field *N* when there are fewer than *N* fields in the case'd constructor.
fetch invalid source	Bit sequence does not correspond to a valid source.
apply arguments to literal	Treating a literal value as a function and passing arguments to it.
apply arguments to constructor	Treating a saturated constructor as a function and passing arguments to it.
application with too many args	Passing more arguments than a function can handle, even if it returns other functions.
application on invalid source	Invalid source designation for function in application.
oversaturated error closure	Passing arguments to an error closure.
oversaturated primitive	Passing more arguments than a primitive operation can handle.
passing non-literal into primitive op	Passing an object (constructor or closure) into a primitive operation.
case on undersaturated closure	Trying to branch on the result of a function that cannot be evaluated.
unused arguments on stack	Oversaturating a function and branching on the result when not all arguments have been consumed.
matching a literal instead of a pattern	Branching on a function that returns a constructor, but trying to match an integer.
invalid skip on literal match	Instruction says to skip *N* words on failed match, but that location is not a branch head.
no else branch on literal match	Incomplete case statement because of lack of else branch.
matching a pattern instead of a literal	Branching on a function that returns an integer, but trying to match a constructor.
incomplete constructor set in case statement	Incomplete case statement because not all possible constructors are present.
invalid skip on pattern match	Instruction says to skip *N* words on a failed match, but that location is not a branch head.
no else branch on pattern match	Incomplete case statement because of lack of else branch.

Table 1: Summary of 21 conditions that require dynamic checks in the absence of static type checking. With our approach, checking is achieved ahead of time, in a single pass through the program; energy and time are not wasted with repeated error checking. No information needs to be tracked at runtime, and the only runtime hardware check is for out-of-memory errors. All of the listed errors are guaranteed by our type system to not occur.

checks is that once the binaries are analyzed, no additional energy and time costs are required during execution. For an embedded system that runs the same code continuously, any small static cost is amortized rather quickly. As we will show later, in fact the static analysis can actually be done in a *single streaming pass* over the executable. However, just to see the scope of the problem it is useful to enumerate some of the dynamic checks that would be required to achieve the same objective as our hardware static analysis.

Table 1 lists ways that programs can fail and costs that are incurred if one were to dynamically check for errors on the platform. There are 21 different ways for the hardware to throw errors, the great majority of which require keeping some significant bookkeeping to actually check. At the very least, we would need to keep extra information on number of arguments, number of local variables, number of recently cased constructor fields, and runtime tags on heap objects to distinguish between closures and constructors — all of which the hardware would need to track at runtime. Crucially, this information must be incorruptible and inaccessible to the software for the dynamic checks to be sound. If software is able to access and corrupt this information, it compromises the integrity of the dynamic checks. In general, guaranteeing that the set of dynamic checks are always occurring, i.e. not bypassed, can be very difficult. With a hardware-implemented static analysis, we are able to formally prove that our checks cannot be bypassed (outlined in Section 6). In addition to the hardware implementation overhead of these checks, reasoning about software behavior in the face of dynamic checks becomes more difficult as well if error states are returned. Programmers that wish to handle errors due to code that fails such checks are forced to reason about every situation that can arise (e.g. what if this function encounters an oversaturated primitive, or cases on an undersaturated closure, and so on.). Instead, by performing the checks statically, all software components understand that any other component with which they might interact on the system is subject to the same analysis as their own code.

2.2 The Bouncer Architecture

Given that we can develop a unit to actually perform the desired static analysis, a big question is where it fits into the actual microcontroller design. Figure 1 shows how a static analysis engine (the Binary Exclusion Unit) fits into an embedded system at a high level: all incoming programs are vetted by the checker before being written to program storage, ensuring that all code that the core executes conforms to the type system's proven high-level guarantees. During programming mode, as a binary image is loaded into the core, the checker has write access to the program store and can use data memory as a working space. The Binary Exclusion Unit can thus be used as a runtime guard, checking programs right before execution when they are loaded into memory, or as a program-time guard, checking programs when they are placed into program storage (flash, NVM, etc.).

Only once the programming mode is complete do the instruction and data memory become visible. The upshot of catching errors this way is that this gives you feedback at programming time, before a device is deployed, that the binary contains errors. It further ensures that when reprogramming occurs in the field, malicious or malformed code that exploits interactions outside of the ISA semantics will never be loaded.

In either case, checking works the same way: each word of the binary is examined one at a time in a linear pass over the program as it is fed through the Binary Exclusion Unit. It is trivial to verify that the BEU is the only unit given access to write to the memory

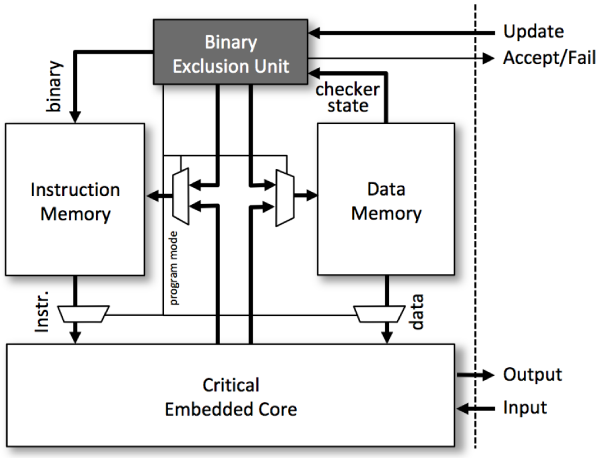

Figure 1: The Binary Exclusion Unit works as a gatekeeper, only allowing input binary programs if they pass the static analysis. When in "Programming" mode, the core is halted while the program is fed to the checker; if it passes, it is written to the system instruction memory. The checker makes use of the core's data memory, which is otherwise unused during system programming. At runtime, the checker is disabled and consumes no resources. Programs that pass static analysis are guaranteed to be free of memory errors, type errors, and control flow vulnerabilities. The checker is non-bypassable; all input binaries are subject to the inspection.

$$
\begin{aligned}
&x \in \mathit{Variable} \quad n \in \mathbb{Z} \quad \mathit{fn}, \mathit{tn}, \mathit{cn} \in \mathit{Name} \quad \oplus \in \mathit{PrimOp} \\
&\alpha \in \mathit{GenericTypeVariable} \quad \beta \in \mathit{RigidTypeVariable} \\
&\mathit{P} \in \mathit{Program} ::= \overrightarrow{\mathit{data}}\ \overrightarrow{\mathit{func}} \\
&\mathit{data} \in \mathit{Datatype} ::= \textbf{data}\ \mathit{tn}\ \vec{\alpha}\ = \overrightarrow{\mathit{cons}} \\
&\mathit{cons} \in \mathit{Constructor} ::= \textbf{con}\ \mathit{cn}\ \vec{\tau} \\
&\mathit{func} \in \mathit{Function} ::= \textbf{fun}\ \mathit{fn}\ \overrightarrow{x : \tau}\ \tau\ = e \\
&e \in \mathit{Expression} ::= \mathit{let} \mid \mathit{case} \mid \mathit{res} \\
&\mathit{let} \in \mathit{Let} ::= \textbf{let}\ x = n\ \textbf{in}\ e \mid \textbf{let}\ x = \mathit{id}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \\
&\mathit{case} \in \mathit{Case} ::= \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br} \mid \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e \\
&\mathit{res} \in \mathit{Result} ::= \textbf{result}\ \mathit{arg} \\
&\mathit{br} \in \mathit{Branch} ::= \mathit{cn}\ \vec{x} \Rightarrow e \mid n \Rightarrow e \\
&\mathit{id} \in \mathit{Identifier} ::= \mathit{fn} \mid \mathit{cn} \mid \oplus \mid x \\
&\mathit{arg} \in \mathit{Argument} ::= n \mid x \\
&\tau \in \mathit{Type} ::= \textbf{Int} \mid \mathit{dt} \mid \mathit{ft} \mid T \\
&\mathit{dt} \in \mathit{Datatype} ::= \mathit{tn}\ \vec{\tau} \\
&\mathit{ft} \in \mathit{FuncType} ::= \vec{\tau} \rightarrow \tau \\
&T \in \mathit{TypeVar} ::= \alpha \mid \beta
\end{aligned}
$$

Figure 2: Typed Zarf Abstract Syntax. An arrow over any metavariable signifies a list of zero or more elements, except for a datatype's constructor list, which must be non-empty.

— the more interesting discussion, covered later, is the verification that the only way though the BEU is via a static analysis.

3 STATIC ANALYSIS STRATEGY

While many different static analysis approaches might be implemented in hardware in the way we described in the sections above, to embody these ideas in a hardware prototype we need a specific analysis specification and implementation. Here we draw inspiration from TAL [22], and use types to clearly and completely specify allowed behavior. By extending the Zarf ISA with types, passing a portion of that type information along with the binary, and then performing the static analysis to check those types, we know the program conforms to the allowed behaviors. This new type-extended Zarf ISA is, unlike untyped Zarf, based on the polymorphic lambda calculus. Figure 2 describes the abstract syntax of the typed ISA; note that there are four types: integers, functions, type variables, and datatypes (which are similar to algebraic datatypes found in languages like Haskell and ML). Both functions and datatypes are declared at the top level; since the ISA is lambda-lifted, the introduction of universally-quantified type variables ranging over a function body or datatype is limited to the top level as well, simplifying the ISA's type system.

Our static analysis requires that type information be encoded into the binary, but we note specifically that the Binary Exclusion Unit discards these annotations when finished, leaving a (safe and certified) standard binary program in protected core memory. To qualify as a typed Zarf program, a binary must declare types of all top-level functions and make all (data) constructors members of a datatype. With this, all types will be tracked and checked, including type variables for polymorphism, facilitating local type inference within the bodies of functions.

The type system in Figure 3 describes, using a set of inference rules, what it means for a Zarf binary to well-typed. Note that the type returned by `applyType` is the principal type of the variable to which it is bound in the `Let` instruction; no constraints are propagated to any instructions that follow, limiting the amount of information that needs to be tracked throughout typechecking, as well as making error reporting of ill-typed applications more accurate.

3.1 Properties and Proofs

Two formal properties, when combined, can guarantee that the machine never has to create and return an error object. The first is progress, which says that if a term is well-typed, then there is always a way to continue evaluating it according to the semantic rules; the second is preservation, which says that if a term is well-typed, evaluating it will result in a well-typed term. Taken together, we have a guarantee that there will always be an applicable semantic rule to evaluate each step of the program, which means that we never encounter anything outside of our semantic definitions and never run into type or memory errors.

We prove progress and preservation in a straightforward way, via induction on the typing rules and the dynamic semantics, giving a brief overview below.

Lemma 3.1 (Apply Type). `applyType` $(\tau_i, \vec{\tau}_a, C, \alpha)$ *returns the principal type of an application of a type to zero or more arguments.*

$$\Gamma \in Env = Variable \to Type \quad C \in ConstraintSet = \mathcal{P}(Type \times Type) \quad \sigma \in Substitution = TypeVar \to Type \quad b \in Bool = \textbf{true} + \textbf{false}$$

Functions $\boxed{\vdash func : \tau}$

$$\frac{\tau_{r1} = \texttt{makeRigid}(\tau_r) \quad \vdash e : \tau_{r2} \quad \texttt{princType}([\tau_{r1}, \tau_{r2}]) = \tau_{r1}}{\vdash \textbf{fun}\ fn\ []\ \tau_r\ =\ e : \tau_r}\ (\textsc{func-ret}) \qquad \frac{(\vec{\tau}_{p1} \to \tau_{r1}) = \texttt{makeRigid}(\vec{\tau}_p \to \tau_r) \quad \vec{x} \mapsto \vec{\tau}_{p1} \vdash e : \tau_{r2} \quad \texttt{princType}([\tau_{r1}, \tau_{r2}]) = \tau_{r1}}{\vdash \textbf{fun}\ fn\ \overrightarrow{x : \tau_p}\ \tau_r\ =\ e : \vec{\tau}_p \to \tau_r}\ (\textsc{func-params})$$

Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\texttt{idTy}(\Gamma, id) = \tau_i \quad \alpha = \texttt{freshGenTV} \quad \Gamma_1 = \Gamma[x_1 \mapsto \alpha] \quad \texttt{map}(\texttt{argTy}(\Gamma_1), \overrightarrow{arg}) = \vec{\tau}_a \quad \texttt{applyType}(\tau_i, \vec{\tau}_a, [], \alpha) = \tau_1 \quad \Gamma[x_1 \mapsto \tau_1] \vdash e : \tau}{\Gamma \vdash \textbf{let}\ x_1 = id\ \overrightarrow{arg}\ \textbf{in}\ e : \tau}\ (\textsc{let-var}) \qquad \frac{\Gamma[x \mapsto \textbf{Int}] \vdash e : \tau}{\Gamma \vdash \textbf{let}\ x = n\ \textbf{in}\ e : \tau}\ (\textsc{let-int})$$

$$\frac{\Gamma(x) = dt \quad \overrightarrow{cons} = \texttt{getCons}(dt) \quad \texttt{allConsPres}(\overrightarrow{cons}, \overrightarrow{br}) = \textbf{true} \quad \vec{\tau} = \texttt{brTypes}(\Gamma, \overrightarrow{br}, \overrightarrow{cons}) \quad \texttt{princType}(\vec{\tau}) = \tau}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br} : \tau}\ (\textsc{case-con}) \qquad \frac{\Gamma(x) = dt \quad \overrightarrow{cons} = \texttt{getCons}(dt) \quad \vec{\tau} = \texttt{brTypes}(\Gamma, \overrightarrow{br}, \overrightarrow{cons}) \quad \Gamma \vdash e : \tau_e \quad \texttt{princType}(\tau_e :: \vec{\tau}) = \tau}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e : \tau}\ (\textsc{case-con-else})$$

$$\frac{\Gamma(x) = \textbf{Int} \quad \overrightarrow{(n_i \Rightarrow e_i)} \in \overrightarrow{br} \quad \Gamma \vdash e_i : \tau_i \quad \Gamma \vdash e : \tau_e \quad \texttt{princType}(\tau_e :: \vec{\tau}_i) = \tau}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e : \tau}\ (\textsc{case-int}) \qquad \frac{\texttt{argTy}(\Gamma, arg) = \tau}{\Gamma \vdash \textbf{result}\ arg : \tau}\ (\textsc{result})$$

$\texttt{applyType} \in Type \times \overrightarrow{Type} \times ConstraintSet \times TypeVar \to Type$

$\texttt{applyType}(\tau_1, \vec{\tau}_a, C, \alpha) =$

$$\begin{cases} \tau_2 & \text{if } \vec{\tau}_a = [] \\ \texttt{applyHelper}(\vec{\tau}_p \to \tau_r, \vec{\tau}_a, C, \alpha) & \text{if } \vec{\tau}_a \neq [] \wedge \tau_1 = \vec{\tau}_p \to \tau_r \end{cases}$$

where

$\sigma = \texttt{unify}(C)$

$\tau_2 = \texttt{substitute}(\sigma, \tau_1)$

$\textbf{true} = (\alpha \notin dom(\sigma)) \vee ((\alpha \mapsto \tau_\alpha) \in \sigma \wedge \texttt{substitute}(\sigma, \tau_\alpha) = \tau_2)$

$\texttt{applyHelper} \in FuncType \times \overrightarrow{Type} \times ConstraintSet \times TypeVar \to Type$

$\texttt{applyHelper}(\tau_p :: \vec{\tau}_p \to \tau_r, \tau_a :: \vec{\tau}_a, C_1, \alpha) =$

$\texttt{applyType}(\tau_f, \vec{\tau}_a, C_2, \alpha)$

where

$C_2 = \{\tau_p = \tau_a\} \cup C_1$

$$\tau_f = \begin{cases} \tau_r & \text{if } \vec{\tau}_p = [] \\ \vec{\tau}_p \to \tau_r & \text{otherwise} \end{cases}$$

Figure 3: Zarf Static Semantics (Typing Rules). See Figure 2 for the abstract syntax. Descriptions of each rule and helper function follow: FUNC-RET checks each function with zero parameters and compares the body's type to the expected return type. FUNC-PARAMS checks functions with parameters; it maps the parameters to their declared types before checking the function body. `makeRigid` universally quantifies all type variables in the type declaration across the body. LET-VAR applies a type to zero or more arguments using the helper `applyType` to get the principal type of the application. Functions may be partially-applied, and mapping the bound variable to a fresh type variable allows for recursive definitions. It checks the next expression in an updated environment. LET-INT performs constant assignment. CASE-CON is used when scrutinizing a datatype; it gets the list of constructors associated with the datatype, replacing all type variables in those constructors' fields with any type variable instantiations found in the datatype, using the helper `brTypes` to get the type of each branch. CASE-CON-ELSE is similar to CASE-CON, but used when all constructors of the datatype aren't present; in this case, an else branch is required. RESULT is the base case, simply producing the type of a bound variable or integer. `applyType` performs constraint generation, unification (`unify`), and substitution (σ) to get the principal type of an application. When no arguments are applied, the type of this helper's first parameter is returned, thus allowing the `Let` instruction to apply an integer, datatype, or generic type as well as function types. `applyHelper` generates constraints between a function application's parameters and arguments, taking care to handle over-application appropriately. `idTy` allows let-polymorphism by replacing non-rigid type variables with fresh ones. `argTy` gets the type of an integer or variable; all arguments must be literals or previously bound in the environment. `brTypes` typechecks a list of branch bodies, mapping the branch's binders to the matching constructor's field types. `princType` verifies a list of types can be considered equal, in the presence of type variables.

Proof. `applyType` generates a constraint for each parameter and argument until the list of arguments $\vec{\tau}_a$ is exhausted. Unifying these constraints to produce a substitution, it then determines the principal type of the application; this proof relies on standard proofs on principal type generation.

Lemma 3.2 (Progress of Functions). *Assuming the correct arguments are given, executing the body of a well-typed function* ***fun*** *$fn\ \overrightarrow{x : \tau}\ \tau\ = e$ produces a value of type $\tau \neq$ `Error`, when the body terminates.*

Proof. The rule FUNC-PARAMS checks a function body e in an environment Γ that maps the parameters in $\vec{x}$ to their declared types in $\vec{\tau}$. Any type variables in those parameter types are universally quantified across the entire function (rule FUNC-RET follows similarly, except that the initial environment used to typecheck the

body e is empty). The proof shows $\Gamma \vdash e : \tau$, that is, that the function body evaluates to a non-error value of type τ, by induction on the derivation of e and using Lemma 3.1.

Theorem 3.3 (Progress of Programs). *Let P be a well-typed program composed of a list of datatypes $\overrightarrow{data}$ and functions $\overrightarrow{func}$. Let $\left(\textbf{fun}\ \texttt{main}\ \overrightarrow{x:\tau}\ \tau\ =e\right) \in \overrightarrow{func}$ be the entry point to P where execution begins. Then P either halts and returns a value of type $\tau \neq \texttt{Error}$, or it continues execution indefinitely.*

Proof. By Lemma 3.2 and rule func-params, we know that **fun** `main` $\overrightarrow{x:\tau}\ \tau\ = e$ has type τ (similarly for functions without parameters, using rule func-ret). Since a hardware error value of type `Error` is created when the machine encounters an invalid state during evaluation, and Lemma 3.2 says that a well-typed function does not lead to an invalid state, P returns a value of type $\tau \neq$ `Error` when it terminates.

4 ALGORITHM FOR ANALYSIS

As mentioned earlier, the Binary Exclusion Unit (BEU) can be used as a runtime guard, checking programs right before execution when they are loaded into memory, or as a program-time guard, checking programs when they are placed into program storage (flash, NVM, etc.). In either case, checking works the same way: each word of the binary is examined one at a time as it streams through. Central to this process is the embedded Type Reference Table (TRT), which is copied from the binary into the checker's memory and contains the type information for the binary. This serves as a reference during all stages of the checking process and will be extended during the checking of each function as local variables are introduced. Later, when the BEU arrives at a new function, it consults the function signature, which provides type information for the arguments and the return type of the function. Each instruction in the function is then scanned word-by-word, guaranteeing type safety of each instruction according to the static semantics (Figure 3). Checking can fail at any step of the process: e.g., a function might expect an `Integer` but is passed a `List`, or the `add` function, which expects two `Integer` arguments, is given three. A single type violation causes the entire program to be rejected. The steps required to check each instruction class are described in more detail below:

Let — When a `Let` instruction is encountered, we first check for special-case operations: applying no arguments to something will always result in the same thing, so we can simply assign the result to that type and do no further checking. Assuming the `Let` does have arguments, the checker then gets the type of the function and creates an alias of it in a new TRT entry. The point of the alias is to make each type variable unique — e.g., the same type `List a` (a list of elements of type "a") used in two places may not be using the same type for "a", so the separate usages should have separate type variables. In order to allow recursive `Let` operations, a type variable is assigned to the result of the operation; when all the arguments have been processed, that variable will be set equal to what's left. The checker goes through each argument, one at a time, and unifies its type with the function's expected type. This creates a list of constraints that, along with the constraint on the resulting type variable, are checked altogether as the last step. If there are no inconsistencies in the constraint set, the operation was valid, and a new valid type is produced for the local variable.

Because type inference is relatively simple, we chose to forgo type annotations on each function application that indicate the result of the operation. Instead, the checker uses function-local type inference to figure out the return type of each function application. Because function calls (`Let` instructions) make up the majority of the instructions in a binary, the absence of annotations on each one results in much smaller binary sizes for typed binaries.

Special care must be taken in `Let` instructions when the resulting type is a function, and when the function being applied has a function in its return type. The former requires creating a new TRT entry for the function; the latter requires a special "unfolding" routine to begin applying arguments to the function in the return type. Both of these are reasons that the `Let` section of the hardware checker has so many states (Table 2).

Case — `Case` instructions are much more straightforward. The checker simply saves some type information on what the program is casing on, which is used in later instructions. Specifically, the primary task is to get the type of the scrutinee (the thing being cased on) and save a reference both to the particular variable's type and the root program datatype (assuming the variable is a constructor, not an integer). For example, this way branches will know that a `List` was cased on, not a `Tuple`, and know that the particular variable was a `List Int` as opposed to a `List Char`.

`Pattern_literal` branch heads are quite simple: the case head must be an integer, and the value specified in the instruction must be an integer.

`Pattern_con` branch heads are one of the more complex things to check. We have to reconcile the generic type of the indicated pattern (constructor) with the specific type of the variable that we're matching against. To do this, the checker must get the function type specified in the pattern head, then alias it in a new TRT entry. Then we must generate the constraint that the return type of the function is the same as the type of the scrutinee — this ensures that the type variables in this entry will be constrained to be the same as those in the original scrutinee. Constraints can then be checked, yielding a map with which the variables can be recursively replaced to the correct types. Finally, a pointer is set to where the fields of the constructor begin (if applicable). When we are done, we have direct, usable information on the type of each field in the constructor, which can be used by following instructions.

In addition, we must keep track of which constructors we've seen in this case statement; that way, when we get to the end of the `Case`, we'll know if all of the constructors of that type were present or not. A `Case` statement must either contain an `else` branch or use all constructors of the scrutinee's type.

5 BEU IMPLEMENTATION

At a high level, the BEU is a hardware implementation of a push-down automaton (PDA) and is structured as a state-machine with explicit support for subroutine calls. While there a numerous book-keeping structures required, we must take care to access a single structure at a time to ensure we do not create structural hazards. The final analysis hardware is the result of a chain of successive lowerings from a high-level static semantics ending with a concrete

(a)

```
data List[a] = Cons a List[a] | Nil

fun map :: ((a) -> b, List[a]) -> List[b]
fun map f list =
case list of {
  Nil => let ret = Nil in ret
  Cons x xs =>
    let head = f x in
    let tail = map f xs in
    let ret = Cons head tail in
    ret
}
```

(b)

```
0 0x40000000   Error
1 0x40000000   Int
2 0x40010002   List a (1 TV, 2 constructors)
3 0xa0000002   Function of 1 arg
4 0xc0000001   arg: Int
5 0xc0000001   return type: Int
6 0xa0000003   Function of 2 args
7 0xc0000001   arg: Int
8 0xc0000001   arg: int
9 0xc0000001   return type: Int
10 0x80000401  Derived Type on List (List a)
11 0xe0000000  arg: Type variable 0
12 0xa0000003  Function of 2 args (Cons)
13 0xe0000000  arg: Typevar 0 (a)
14 0xc000000a  arg: List a
15 0xc000000a  return type: List a
16 0xa0000002  Function of 1 arg (a -> b)
17 0xe0000000  arg: Typevar 0
18 0xe0000001  arg: Typevar 1
19 0x80000401  Derived Type on List (List b)
20 0xe0000001  arg: Type variable 1
               Function Signatures:
25 0xa0000001  Function of no args (main)
26 0xc0000001  return type: Int
27 0xa0000003  Function of 2 args (Cons)
28 0xe0000000  arg: Typevar 0
29 0xc000000a  arg: Reference to 10
30 0xc000000a  return type: Reference to 10
31 0xa0000001  Function of no args (Nil)
32 0xc000000a  return type: Reference to 10
33 0xa0000003  Function of 2 args (map)
34 0xc0000010  arg: Reference to 16
35 0xc000000a  arg: Reference to 10
36 0xc0000013  return type: Reference to 19
```

Figure 4: An example Type Reference Table (TRT) for the function `map`. The original program is shown in (a), while (b) shows the actual binary type information that the assembler produces (annotated for human understanding). This type information is included at the head of the binary file, leaving the program unchanged. The first section lists types used in the signatures of the program, while the second section contains type information for the parameters and return type of each function. The type system is polymorphic and uses function-local type inference.

state chart that we could then implement with minimal and straighforward hardware. First, a bit-accurate software checker was made that checked binary files. Then, a cycle-accurate software pushdown automata was written from that refined specification. From that program, the leap to real hardware was somewhat straightforward (see Section 7 for synthesis results). The full details of the checker cannot hope to fit in this paper, so we concentrate here on the strategy used at a high level and a couple of details to give a better sense for the full design.

The first challenge in implementing this analysis is how to encode the type information into the binary. As discussed in the prior section, we put this information at the head of each binary in the form of the TRT. To get a sense of what that actually looks like in a real implementation, Figure 4 shows an example TRT for the function `map`. This information is discarded after checking, leaving a standard, untyped binary, which executes with normal performance.

At the bit-level, we see only a sequential series of bytes. Therefore, all type information must be encoded into a single list. To avoid unnecessary complexity, we make all entries in the TRT fixed-width 32-bit words. An entry can be either 1) a program-specified datatype or built-in type[1], or 2) a derived type based on another type. Entries of type 2 can have one or more argument words, which we refer to as "typewords." "Derived" here means that the entry contains references to other types in the table. This manifests as either a type applied to some type variables or as a function. For example, `List` is specified as a program datatype with one type variable, then derived type entries can create the types `List a`, `List Int`, etc, where `a` and `Int` are typewords following the derived type entry.

The second challenge in bringing the typechecker to a low level is dealing with recursive types. Implicitly, types in the system may be arbitrarily nested: for example, one could declare a `List` of `Tuples` of `Lists` of `Ints`. During the checking process, the hardware typechecker must be able to recursively descend through a type in order to make copies, do comparisons, and validate types. Because of this, the Binary Exclusion Unit cannot be expressed as a simple state machine — a stack is required for recursive operations (and hence the pushdown automaton).

Data structures used in the higher-level checking, like maps, need to be converted to structures native to hardware: they must flattened into a list, which can be stored in memory. In some cases, this requires a linear scan to check for the presence of some elements, such as checking case completeness — but those lists tend to be small, containing just one entry for each constructor of a given datatype. We found that all of the structures could be represented as lists with stack pointers, except in the case of the type variable map used in the recursive replace procedure, which required two lists (one to check for membership in the map, the second with values at the appropriate indices).

To create the control structure of the PDA, we started by implementing a software-level checker, broken into a set of functions implemented with discrete steps, where each step cannot access more than one array in sequence (in hardware, the arrays will become memories, which we do not want strung together in a single cycle). While, given our space constraints, it is difficult to describe the system in detail, the number of states for each part of the analysis is a reasonable proxy for complexity. The resulting state machine has 207 states and they are broken down by purpose in Table 2. We summarize them briefly here, with number of states denoted in parentheses. The initialization stage reads the program and prepares the type table (21 states). Function heads are checked to ensure the argument count matches the provided function signature, and bookkeeping is done to note the types of each argument and the return type (15). Dispatch decides which instruction is executed next and handles saving and restoring state as necessary for `Case` statements (6). `Let` (37), `Result` (3), `Case` (7), `Pattern_literal` (1), and `Pattern_con` (21) are checked as outlined in Section 4.

Because types can be recursively nested, a type entry in the TRT can reference other types; a set of states is devoted to following references to find root types as needed (6 states). To handle this, the state machine implements something akin to subroutines. A

[1]The Zarf ISA includes integers and an error datatype built-in.

Purpose	Number of States
Initialization	21
Function signatures	15
Dispatch	6
Let checking	37
Return checking	3
Case checking	7
Literal pattern checking	1
Constructor pattern checking	21
Following references	6
Type variable (TV) counting	12
Recursive TV replacement	12
Recursive TV aliasing	26
Generating constraints	19
Checking constraints	21
Total	**207**

Table 2: Number of states devoted to the various parts of the Binary Exclusion Unit's state machine. Checking function calls, allowing for polymorphic functions with type variables, and constraint checking were the most complex behaviors, making up most of the states.

routine executes at the beginning of each function that counts the number of type variables used in the signature (these type variables are "rigid" within the scope of the function and cannot be forced to equal a concrete type) (12). Another routine recursively replaces type variables to make one type entry match the variables in another; it allows pattern heads to be forced to the same type as the variable in the `Case` instruction (12). The aliasing subroutine recursively walks a type and maps its type variables to a "fresh" set (26). This allows, for example, each usage of `List a` to have "`a`" refer to a different type. Part of the complexity of this task is keeping track of the variables already seen and what they map to so that a variable is not accidentally mapped twice to different values. Constraint generation takes two type entries and, based on the entries, branches and generates the appropriate constraint for the constraint set indicating that the entries should be equal (19).

Finally, we have the constraint checking routine (21). This is invoked at the end of each `Let` instruction, as well as after a `Result`. Constraint propagation proceeds by taking one constraint from the set, which consists of a pair of types, then walking all the remaining constraints in the set and replacing all occurrences of the first type with the second. In this way, for each unique constraint, one type is eliminated from the constraint set. If at some point two different concrete types (like `Int` and `List`) are found to be equal, the set is inconsistent and typechecking fails, rejecting the program. Similarly, if ever a rigid type variable (a type variable used in the function signature) is found to be equal to a concrete type, typechecking fails. This second fail condition ensures that functions with polymorphic type signatures are, in fact, polymorphic. Without it, one could write a function that takes "`a`" and returns "`a`", which *should* work for all types, but in fact only works for integers.

As we developed our software and hardware checkers, we used a software fuzzing technique to generate 200,184 test cases based on prior techniques in program testing [15]. Rather than generating random bits, which would not meaningfully exercise the checker, we encode the type system's semantics with logic programming and run them "in reverse" to generate, rather than check, well-typed programs. By performing mutations on half of these programs, we also generate ill-typed benchmarks. In all 200,184 generated test cases, the simulated hardware RTL has 100% agreement with the high-level checker semantics. The tests provide 100% of coverage of all 207 states of the checker.

While the resulting analysis engine is complex, one could certainly reuse parts of the analysis for other sets of properties and automated translation would be an interesting direction for future work. The software model is 1,593 lines of Python, while the hardware RTL is 1,786 lines (requiring extra code for declarations and the simulation test harness). Synthesis results are found in Section 7.

6 PROVABLE NON-BYPASSABILITY

The hardware static analysis we developed has a variety of states governing when it is active, how it initializes, and so on. An important point of this paper is the non-bypassibility of these checks, but we need to know that some sequence of inputs cannot be given to the checker that causes outputs to write to memory that have not been checked by analysis. To solve this problem, we can create an assertion and employ the Z3 SMT solver [13] to check it for us. Z3 is well-suited to our task because of its ability to represent logical constructs and solve propositional queries. In addition, because we can directly represent the circuit in Z3 at the logic level (gates), we do not have to operate at higher levels of abstraction and risk the proof not holding for the real hardware.

We actually translate our entire analysis circuit into a large Z3 expression. Then, we add two constraints: the first says that, at some point in the operation of the circuit, it output the "passed" (meaning well-typed) signal, while the second says that at no point did the hardware enter the checking states. If the conjunction of the expressions is unsatisfiable, there is no way to get a "pass" signal without undergoing checking (and the program will never be loaded if it fails checking). Around 30 of the states deal with program loading, initialization, etc., and perform no checking; our proof guards against, for example, situations in which some clever manipulation of the state machine moves it from initialization directly to passing, or otherwise manages to circumvent the checking behavior of the state machine.

In the most direct strategy, we use the built-in `bitvec` Z3 type for wires in the circuit, with gates acting as logical operations on those bitvectors. Memories are represented as arrays. Arrays in Z3 are unbounded, but because we address the array with a bitvector, there is an implicit bound enforced that makes the practical array non-infinite.

A straightforward approach to handling sequential operation of the analysis is to duplicate the circuit once for each cycle we wish to explore. The cycle number is appended to the name of each variable to ensure they are unique. Obviously, because the entire circuit is duplicated for each cycle, this method does not scale well — both in terms of memory usage and the time it takes to determine satisfiability. Checking non-bypassability for numbers of cycles up to 32 took under 2 minutes and used less than 1 GB of RAM.

Checking for 64 cycles used almost 16 GB and did not terminate within four days.

To make the SMT query approach scalable, we employ Z3's theory of arrays. Instead of representing each wire as a bitvector, duplicated once for each cycle, we represent it as an array mapping integers to bitvectors: the integer index indicates the cycle, while the value at the index is the value the wire takes in that cycle. There is then one array for each wire in the circuit, and one array of arrays for each memory in the circuit (the first array represents the memory in each cycle, while the internal array gives the state of the memory in that cycle). Logical expression (gates) can then be represented as *universal quantifiers* over the arrays. For example, an AND expression might look like, `ForAll(i, wire3[i] == wire1[i] & wire2[i])`. This constrains the value of `wire3` for all cycles. Sequential operations are easy, simply referring to the previous index where necessary for register operations, e.g. `ForAll(i, reg1[i] == reg1_input[i-1])`. To bound the number of cycles, we add constraints to each universal quantifier that `i` is always less than the bound; this prevents Z3 from trying to reason about the circuit for steps beyond `i`.

Solving satisfiability with arrays took under two minutes and under one GB of RAM, no matter what bound we placed on the cycle count — in fact, even when *unbounded*, Z3 was still able to demonstrate our hardware analysis bypassibility assertion was unsatisfiable — i.e., the circuit is non-bypassable.

7 EVALUATION

7.1 Checking Benchmarks

To understand if real-world programs can be efficiently typed and checked with our system, we implement a subset of the benchmarks from MiBench [17]. These tended to be much longer and more complex programs when compared to the randomly-generated ones. While the fuzzer's programs averaged 50-65 instructions per program, the embedded benchmarks range from 500 to over 7,000 and represent code structures drawn from real-world applications, such as hashes, error detection, sorting, and IP lookup. In addition to the MiBench programs, a standard library of functions was checked, as well as a synthetic program combining all the other programs (to see the characteristics of longer programs).

Figure 5 shows how long typechecking took for the benchmark programs as a function of their code size. A linear trend is clearly visible for most of the programs, but one stands out from the pack: the CRC32 error detection function. The default CRC32 implementation is, in fact, a pathological case for our checking method as it is dominated by a single large function in the program. This function constructs a lookup table used elsewhere and is fully unrolled in the code. No other benchmark had a function nearly as large. The typecheck algorithm, while linear in program length (it checks in a single pass), is quadratic in function length and type complexity[2]. This insight not only explains the anomalous behavior of the initial CRC32 program, but provides a clear solution: break up the large function.

We test this hypothesis by breaking up CRC32 and re-checking it. While the task of breaking up a function in a traditional imperative programming language is complicated by the large amounts of global and implicit state, and would be even harder to perform at level of assembly, in a pure functional environment every piece of state is explicit. This makes the process not only easier, but even possible to fully automate. When we look at CRC32 specifically, the state, passed directly from one instruction to the next for table composition, can be captured in a single argument. We perform this transformation on our CRC32 program to break table construction across 26 single-argument functions, producing the CRC-short data point in the graphs in Figure 5. It still stands slightly above average because the table-construction functions are still above the average function length; recursively applying the breaking procedure could easily reduce the gap further.

While function length is an important aspect of checking time, with some care it can be effectively managed, and in the end all of the programs examined can be statically analyzed in hardware at a rate greater than 1 instruction per 100 cycles. This rate is more than fast enough to allow checking to happen at software-update-time, and could perhaps even be used at load-time, depending on the sensitivity of the application to startup latency.

7.2 Practical Application to an ICD

In addition to the benchmarks described above, we additionally provide results for a complete embedded medical application that was typed and checked; specifically, an ICD, or implantable cardioverter-defibrillator[3]. The ICD code was the largest single program examined (only the synthetic, combined program was larger). Its complexity required the use of multiple cooperating coroutines, managed by a small microkernel that handled scheduling and communication. Despite its length and complexity, it had the best typecheck characteristics of any of our test programs, with its cycles-per-instruction figure falling just below the average at 55.2. The process of adding types to the application was relatively simple, taking approximately 2 hours by hand.

Since the ICD represents the largest and most complex program, as well as the exact type of program the BEU is designed to protect, we attempt to introduce a set of errors in the program to demonstrate the ability of the BEU to ensure integrity and security. Some of the errors are designed to crash the program; some are designed to hijack control flow; others are designed to read privileged data. The list of attempted attacks and how the BEU caught them are shown in Table 3.

In an unchecked system, passing an invalid function argument, writing past the end of an object, and passing an invalid number of function arguments could all lead to undefined behavior or system crashes. While past work could establish that a specific piece of code would not do these things independent of the device, this work establishes these properties for the device itself, applying to all programs that can potentially execute — it is simply impossible to load a binary that will allow these errors to manifest. To establish that this was indeed the case, Table 3 shows the result of our attempts to produce these behaviors: a type error, a function application error, and an undersaturated call error, respectively. Reading past the end

[2] "Type complexity" refers to how many base types are in a type; i.e., the length of its member types, recursively.

[3] An ICD is a device planted in a patient's chest cavity, which monitors the heart for life-threatening arrhythmias. In the case one is detected, a series of pacing shocks are administered to the heart to restore a safe rhythm.

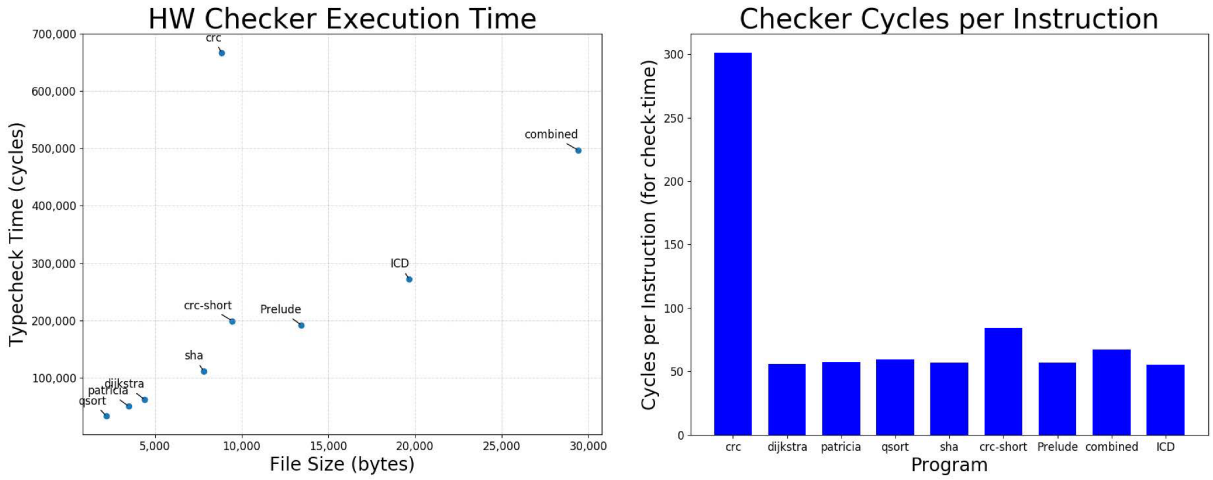

Figure 5: BEU evaluation for a set of sample programs drawn from MiBench, an embedded benchmark suite. For most programs, complete binary checking will take 150-160 cycles per instruction. LEFT: Time for hardware checker to complete, in cycles, as a function of the input program's file size. RIGHT: The same checking time, divided over the number of instructions in each program. Though the stock CRC32 has the longest typecheck time, an automatic procedure can modify the program to lower the checking time while preserving program semantics, noted as CRC-short.

Attempted Attack	Result
Binary that reads past the end of an object to access arbitrary memory	Hardware refuses to load binary due to type error "field count mismatch"
Binary that passes an argument to a function of the wrong type to cause unexpected behavior	Hardware refuses to load binary due to type error "not expected type"
Binary that writes past the end of an object to corrupt memory	Hardware refuses to load binary due to "application on non-function type"
Binary that passes too few arguments to a function to attempt to corrupt the stack	Hardware refuses to load binary due to "undersaturated call"
Binary that uses an invalid branch head to try and make arbitrary jump	Hardware refuses to load binary due to type error "branch type mismatch"
Binary that jumps past the end of a case statement to enable creation of ROP gadgets	Hardware refuses to load binary due to "invalid branch target"
Jump past the end of a function to create ROP gadgets	Hardware refuses to load binary due "invalid branch target"

Table 3: A list of some of the erroneous code that may be present in a binary (tested in our ICD application) and how the BEU identifies it as an error. Some of these errors, such as reading off the end of an object, writing beyond the end of an object, and jumping to arbitrary code points, are sufficient to thwart common attacks, like buffer overflow and ROP.

of an object in an attempt to snoop privileged data was thwarted by detecting a type error dealing with field count mismatches. Control-flow hijacks, like using an invalid branch head, jumping past the end of a case statement, and jumping past the end of a function, were caught by a type mismatch in the first case and the detection of an invalid branch target in the latter two.

Though not exhaustive, these attacks show the resilience of the system to injected errors when compared to an unchecked alternative and demonstrate its practicality in the face of real errors and attempted attacks.

7.3 Synthesis Results

Synthesized with Yosys, the hardware typechecker logic uses 21,285 cells (of which 829 are D Flip Flops, the equivalent of approximately 26 32-bit registers). Mapped to the open-source VSC 130nm library, it is .131 mm^2, with a clock rate of 140.8 MHz. Scaled to 32nm, it is approximately .0079 mm^2. As an addition to an embedded system or SoC, it provides only a tiny increase in chip area, and requires no power at run-time (having already checked the loaded program).

Assuming the checker can use the system memory, it requires no additional memory blocks; if not, it needs a memory space at least as large as the input binary type information, and space linear in the size of the program's functions.

The worst-case checking rate was 301 cycles per instruction for a pathological program; even a program of 450,000 lines with worst-case checking performance can be checked in under a second at the computed clock speed of 140 MHz on 130nm.

8 RELATED WORK

Typed Assembly

When dealing with typed assembly, the most prominent works are TAL [22] and its extensions TALx86 [10], DTAL [27], STAL [21], and TALT [9]. In TAL, they demonstrate the ability to safely convert high-level languages based on System F (e.g. ML) into a typed target assembly language, maintaining type information through the entire compilation process. Their target typed assembly provides several high-level abstractions like integers, tuples, and code labels, as well as type constructors for building new abstractions.

TALx86 is a version of IA32, extending TAL to handle additional basic type constructors (like records and sums), recursive types, arrays, and higher-order type constructors. They use dependent types to better support arrays; the size of an array becomes part of its type, and they introduce singleton types to track integer values of arbitrary registers or memory words. TAL provides a way to

ensure that high-level properties like type- and memory-safety are preserved after compiler transformations and optimizations have taken place.

Unlike TAL, our type system was co-designed with hardware checking in mind — a distinction that greatly impacts the type system design. It allows for binary encoding of types and empowers the target machine, rather than the program authors, to decide if a program is malformed. TAL requires a complex, compile-time software typechecker, as opposed to our small, load-time hardware checker. Our type system operates on an actual machine binary and not an intermediate language.

The eventual target of TALx86 is untyped assembly code (assembled by their MASM assembler into x86). The types are not carried in the binary and are not visible to the device that ultimately runs the code. Though useful, a device cannot trust that the program it has been given has been vetted; therefore, bad binaries can still run on TAL's target machines.

Our work's most significant contribution, the Binary Exclusion Unit (BEU), overcomes this problem. The BEU, a hardware typechecker for the system capable of rejecting malformed programs, is an integral, non-bypassable part of the machine; if typechecking fails, execution cannot begin. To our knowledge, this is the only hardware module that performs typechecking on binary programs. We leave expansion of the BEU to other ISAs for future work, but note that the complexity of the TAL type system indicates that a hardware implementation would be significantly more work and overhead on an imperative ISA.

Architecture and Programming Languages

In SAFE [2], the authors develop a machine design that dynamically tracks types at the hardware level. Using these types along with hardware tags assigned to each word, their system works to prove properties about information-flow control and non-interference. They claim that the generic architecture of their system could facilitate efforts related to memory and control-flow safety in further work.

There has also been important work in binary analysis, which seeks to recover information from arbitrary binaries to make sound and useful observations. For example, Code Surfer [3] is a tool that analyzes executables to observe run-time and memory usage patterns and determine whether a binary may be malicious. Work on binary type reconstruction in particular seeks to recover type information from binaries. In one work [19], they recover high-level C types from binaries via a conservative inference-based algorithm. In Retypd [25], Noonan et al. develop a technique for inferring complex types from binaries, including polymorphic and recursive structures, as well as pointer, subtyping, and type qualifier information. Caballero et al. [5] provide a survey of the many approaches to binary type inference and reconstruction.

Static safety via on-card bytecode verification in a JavaCard [6] is an interesting line of work with a similar goal to our approach. However, a hardware implementation can be verified non-bypassable in a way that is much harder to guarantee for software. The Java type system is known to both violate safety [1, 8] and be undecidable [16] which makes it a far more difficult target for static analysis and, we would argue, nearly impossible to implement in hardware directly.

At the intersection of hardware and functional programming, previous works have synthesized hardware directly from high-level Haskell programs [28], even incorporating pipelined dataflow parallelism [26]. Run-time solutions to help enforce memory management for C programs have been proposed at the software level [24], as well as in hardware-enforced implementations [14, 23]; these provide run-time, rather than static, checks.

Other work has used formal methods to find and enforce properties at the hardware level to help ensure hardware and software security [29], while others have shows the effectiveness of hardware-software co-analysis for exploring and verifying information flow properties in IoT software [7].

9 CONCLUSION

While the micro-controller design in this paper might be an extremely non-traditional example, going so far as to have proofs of the properties that hold and rejecting non-conforming programs outright, it opens the door to other work that limits hardware functionality in meaningful and helpful ways without entirely giving up programmability. The result of our effort is a Binary Exclusion Unit that can easily fit into embedded systems or perhaps even serve as an element in a heterogeneous system-on-chip, providing a hardware-based solution that cannot be circumvented by software. Our approach prevents all malformed binaries from ever being loaded (let alone run), and ensures that all code loaded onto the machine is free from memory errors, type errors, and erroneous control flow. It requires neither special keys/attestation nor trust in any part of the system stack (a size zero TCB), providing its guarantees with static checks alone (no dynamic run-time checking is needed).

This approach has many non-traditional moving parts, from the function-oriented microprocessor at its heart, to the higher-level instruction set semantics, to the engine that performs static analysis in hardware. Rather than work on an architecture simulator, we built both the processor and the hardware checking engine in RTL, both to provide synthesis results and to demonstrate the feasibility of actually building such a thing. We have proofs of correctness for our approach at the algorithm level and gate-level proofs of non-bypassability. We coded and typed not just a set of benchmarks, but also a more complete medical application, which we then tried to break in order to show that such an approach works in practice as well as in theory. The final design is surprisingly small, taking no more than .0079 mm^2, and is capable of performing our static analysis on binaries at an average throughput of around 60 cycles per instruction. We believe this is the first time any binary static analysis has been implemented in the machine hardware, and we think it opens an interesting new door for exploration where properties of the software running on a physical platform are enforced by the platform itself.

Acknowledgements

We'd like to thank the anonymous reviewers for their invaluable feedback in getting this paper to where it is.

This material is based upon work supported by the National Science Foundation under Grants No. 1740352, 1730309, 1717779, 1563935, 1350690, and a gift from Cisco Systems.

REFERENCES

[1] Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 838–848. https://doi.org/10.1145/2983990.2984004

[2] Arthur Azevedo de Amorim, Nathan Collins, AndrÃĬ DeHon, Delphine Demange, CÄČtÄČlin HriÅčcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2014. A Verified Information-flow Architecture. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 165–178. https://doi.org/10.1145/2535838.2535839

[3] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86âĂŤA Platform for Analyzing x86 Executables. In *Compiler Construction (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 250–254. https://doi.org/10.1007/978-3-540-31985-6_19

[4] Erik Buchanan, Ryan Roemer, and Stefan Savage. [n. d.]. Return-Oriented Programming: Exploits Without Code Injection. https://hovav.net/ucsd/talks/blackhat08.html. ([n. d.]).

[5] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *ACM Comput. Surv.* 48, 4 (May 2016), 65:1–65:35. https://doi.org/10.1145/2896499

[6] Zhiqun Chen. 2000. *Java card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional.

[7] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Software-based Gate-level Information Flow Security for IoT Systems. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 328–340. https://doi.org/10.1145/3123939.3123955

[8] Wouter Coekaerts. [n. d.]. The Java Typesystem is Broken. http://wouter.coekaerts.be/2018/java-type-system-broken. ([n. d.]).

[9] Karl Crary. 2003. Toward a Foundational Typed Assembly Language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 198–212. https://doi.org/10.1145/604131.604149

[10] K. Crary, Neal Glew, Dan Grossman, Richard Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. 1999. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*. 25–35. http://www.cis.upenn.edu/~stevez/papers/MCGG99.pdf

[11] Ang Cui, Michael Costello, and Salvatore J. Stolfo. 2013. When Firmware Modification Attack: A Case Study of Embedded Exploitation. In *NDSS Symposium '13*.

[12] Zach Cutlip. 2013. Dlink DIR-815 UPnP Command Injection. http://shadow-file.blogspot.com/2013/02/dlink-dir-815-upnp-command-injection.html. (February 2013). [Online; accessed 01-November-2017].

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[14] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 103–114. https://doi.org/10.1145/1346281.1346295

[15] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 482–493. https://doi.org/10.1109/ASE.2015.65

[16] Radu Grigore. 2016. Java Generics are Turing Complete. *CoRR* abs/1605.05274 (2016). arXiv:1605.05274 http://arxiv.org/abs/1605.05274

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01)*. IEEE Computer Society, Washington, DC, USA, 3–14. https://doi.org/10.1109/WWC.2001.15

[18] Grant Hernandez, Orlando Arias, Daniel Buentello, and Yier Jin. 2014. Smart Nest Thermostat: A Smart Spy in Your Home, In Black Hat Briefings. *Black Hat Briefings*.

[19] Jonghyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. In *In Proceedings of the Network and Distributed System Security Symposium*.

[20] Joseph McMahan, Michael Christensen, Lawton Nichols, Jared Roesch, Sung-Yee Guo, Ben Hardekopf, and Timothy Sherwood. 2017. An Architecture Supporting Formal and Compositional Binary Analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 177–191. https://doi.org/10.1145/3037697.3037733

[21] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-based typed assembly language. *Journal of Functional Programming* 12, 01 (Jan. 2002). https://doi.org/10.1017/S0956796801004178

[22] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.* 21, 3 (May 1999), 527–568. https://doi.org/10.1145/319301.319345

[23] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 189–200. http://dl.acm.org/citation.cfm?id=2337159.2337181

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[25] Matthew Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. *arXiv:1603.05495 [cs]* (March 2016). http://arxiv.org/abs/1603.05495 arXiv: 1603.05495.

[26] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. 2017. From Functional Programs to Pipelined Dataflow Circuits. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. ACM, New York, NY, USA, 76–86. https://doi.org/10.1145/3033019.3033027

[27] Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 169–180. https://doi.org/10.1145/507635.507657

[28] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. 2015. Hardware Synthesis from a Recursive Functional Language. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis (CODES '15)*. IEEE Press, Piscataway, NJ, USA, 83–93. http://dl.acm.org/citation.cfm?id=2830840.2830850

[29] Rui Zhang, Natalie Stanley, Chris Griggs, Andrew Chi, and Cynthia Sturton. 2017. Identifying Security Critical Properties for the Dynamic Verification of a Processor. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.